

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

OBJECT SIGNING IN BAMBOO

by

Marlon L. Smith

March 2000

Thesis Advisor:
Thesis Co-Advisor:

Michael J. Zyda
John S. Falby

Approved for public release; distribution is unlimited.

20000530 047

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE

March 2000

3. REPORT TYPE AND DATES COVERED

Master's Thesis

4. TITLE AND SUBTITLE

Object Signing in Bamboo

5. FUNDING NUMBERS

6. AUTHOR(S)

Marlon L. Smith

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School

Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION
REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING
AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

The rapid growth in the Internet has been fueled by an exorbitant number of users, organizations and individuals alike, many relying on e-commerce to conduct business including the transport of files. Public Key Infrastructure (PKI) technology has emerged to the forefront as the basis for ensuring secure transactions throughout the Internet. However, this technology is prohibitively expensive for the majority of users. Object signing technology, a subset of PKI technology, provides a veritable means for file transfer ensuring non-repudiation, authentication, and content integrity at an amenable cost.

This thesis provides an introduction to computer security with a specific focus on PKI and object signing technology. It details the selection and implementation of an object signing system layered on Bamboo, namely Pretty Good Privacy (PGP) v2.6.2. Procedures for establishing a Key Server for certificate distribution are also illustrated. It also introduces security pitfalls associated with PKI systems and identifies the security weaknesses of this object signing implementation. For further research, recommendations are provided to improve the overall functionality of this security system and the potential impact any such migration may have on current users.

14. SUBJECT TERMS

Object Signing, Public Key Infrastructure, PKI, PGP

15. NUMBER OF PAGES

134

16. PRICE CODE

17. SECURITY
CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF
THIS PAGE
Unclassified

19. SECURITY CLASSIFI-
CATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
UL

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

OBJECT SIGNING IN BAMBOO

Marlon L. Smith
Lieutenant Commander, United States Navy
B.S., Bowling Green State University, 1982

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN MODELING VIRTUAL ENVIRONMENTS
AND SIMULATION**

from the

**NAVAL POSTGRADUATE SCHOOL
March 2000**

Author:

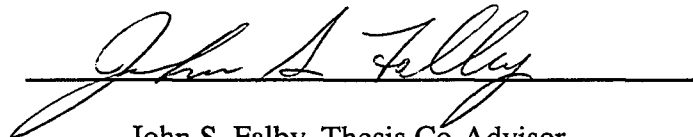


Marlon L. Smith

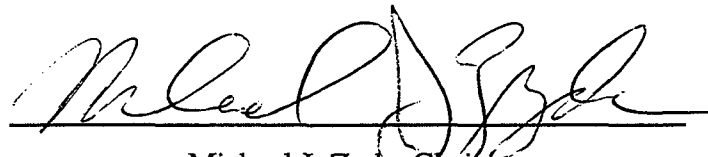
Approved by:



Michael J. Zyda, Thesis Advisor



John S. Falby, Thesis Co-Advisor



Michael J. Zyda, Chair,
MOVES Academic Group

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The rapid growth in the Internet has been fueled by an exorbitant number of users, organizations and individuals alike, relying on e-commerce to conduct business including the transport of files. Public Key Infrastructure (PKI) technology has emerged to the forefront as the basis for ensuring secure transactions throughout the Internet. However, this technology is prohibitively expensive for the majority of users. Object signing technology, a subset of PKI technology, provides a veritable means for file transfer ensuring non-repudiation, authentication, and content integrity at an amenable cost.

This thesis provides an introduction to computer security with a specific focus on PKI and object signing technology. It details the selection and implementation of an object signing system layered on Bamboo, namely Pretty Good Privacy (PGP) v2.6.2. Procedures for establishing a Key Server for certificate distribution are also illustrated. It also introduces security pitfalls associated with PKI systems and identifies the security weaknesses of this object signing implementation. For further research, recommendations are provided to improve the overall functionality of this security system and the potential impact any such migration may have on current users.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I.	INTRODUCTION	1
	A. PROBLEM STATEMENT	1
	B. MOTIVATION	1
	C. THESIS ORGANIZATION	4
II.	BACKGROUND	7
	A. INTERNET SECURITY ISSUES	7
	B. A GENERAL PKI SYSTEM	12
	C. A GENERAL OBJECT SIGNING SYSTEM	21
	D. SUMMARY	27
III.	SELECTION	29
	A. SOFTWARE REQUIREMENTS SPECIFICATION	29
	B. COMMERCIAL SYSTEMS AVAILABLE	37
	C. SELECTING A SYSTEM	40
IV.	IMPLEMENTATION	43
	A. PGP KEYS	43
	B. USING PGP WITH BAMBOO	58
	C. BAMBOO PGP PUBLIC KEY SERVER	69
	D. ESTABLISHING A SECURITY POLICY FOR PUBLIC KEYS	74
V.	RESULTS	77
	A. RISKS IN A TYPICAL PKI SYSTEM	77
	B. RISKS ASSOCIATED WITH PGP v2.6.2	80
	C. RESULTS OF PGP v2.6.2 IMPLEMENTATION IN BAMBOO	83
VI.	CONCLUSIONS AND RECOMMENDATIONS	87
	A. CONCLUSIONS	87
	B. SIGNIFICANCE	87
	C. FUTURE WORK	88
	APPENDIX A. USER'S GUIDE FOR BAMBOO OBJECT SIGNING	93
	APPENDIX B. BAMBOO PGP PUBLIC KEY SERVER ADMINISTRATOR'S GUIDE	103
	LIST OF REFERENCES	121
	INITIAL DISTRIBUTION LIST	123

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGEMENT

The author would like to acknowledge the efforts of a multitude of individuals who provided insight and guidance from the conceptual stage through the implementation of the security application. This thesis recognizes Kent Watsen for his invaluable technical knowledge and formalized security requirements in Bamboo; Professor Don Brutzman for his expertise in Netscape Object Signing technology; and both MOVES Academic Group advisors, Professor John Falby and Professor Mike Zyda for their guidance and patience in the delivery of the final product.

Additionally, this thesis recognizes the previous work of Marc Horowitz [Ref. 20] and utilizes his free implementation of the PGP Public Key Server. It also applauds Network Associates for providing the PGP freeware security product through the MIT website, and recognizes Phil Zimmerman's contributions in the original development of the product and his extensive documentation.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM STATEMENT

With the rapid expansion of the Internet in recent years, numerous companies have emerged with products purported to provide security for electronic commerce in the areas of personal banking, online purchasing, business-to-business transactions as well as various other areas utilizing a typical public key infrastructure (PKI) system. Additional focus has been attached to "object signing" for delivery of files across a network in a manner ensuring a certain degree of content integrity and authentication.

This thesis provides an introduction to computer security with a specific focus on PKI and object signing technology. The major objective of this thesis includes the selection and development of an object signing system implemented in Bamboo, a dynamically extensible, cross-platform, component-based framework for virtual environments under development at the Naval Postgraduate School. Security sub-areas such as physical security, types of encryption and hash algorithms to preclude code breaking, etc., are only briefly covered. However, it is not the intent to undermine the importance of these sub-areas. A security system will only be as effective as its weakest link. This thesis discusses the benefits of object signing in Bamboo; specific focus areas include:

- acceptable cost to an open-source community.
- potential shortcomings in PKI technology and the implemented security system within Bamboo.
- basis for selection of a particular system based on "a priori" criteria.

The research also discusses various object signing systems available and pertinent factors such as ease of implementation, initial and recurring costs, and robustness of such systems.

B. MOTIVATION

1. DOD Relevance

The Department of Defense (DOD) sets forth guidelines pertaining to information security which include the following requirements [Ref. 1]:

- To put in place information assurance solutions needed to assure the availability, integrity, authentication, confidentiality, and non-repudiation of the Defense Information Infrastructure (DII).
- To provide an extended foundation for the integrated, continuous, multi-layered (defense in depth) protection of the Defense Information System Network (DISN), the backbone of the DII.
- To comply with the Deputy Secretary of Defense (DEPSECDEF) and Assistant Secretary of Defense (Command, Control, Communications and Intelligence) memorandums mandating migration to PKI technology to protect sensitive but unclassified information transfer.
- To ensure a government link to the industrial base.

A typical PKI system available in this rapidly emerging field fully meets the requirements set forth above; such a system will be discussed in Chapter II.

2. Benefits of a PKI System

PKI systems may offer an organization numerous benefits during the transmission of vital information across a network. A typical system provides the following assurances [Refs. 1, 2, 3, 4]:

- Authentication: proof that the sender is whom they claim to be (public/private key).
- Non-repudiation: assurance that the person sending cannot deny participation (digital signature).
- Integrity: verification that no unauthorized modification of data has occurred (hash).
- Confidentiality: assurance that the person receiving is the intended recipient (encrypt/decrypt).

3. Benefits of Object Signing

Object signing is an emerging technology with inherent roots from PKI and is used primarily for electronic software distribution. It provides the means for a user to obtain reliable information about downloaded code such as a list of other users who have

digitally signed the code (authentication) and whether the downloaded code has been tampered with (content integrity). Further details are incorporated in Chapter II.

4. Bamboo

To better understand the motivation for selection of a particular object signing implementation, it is necessary to comprehend the underlying Bamboo system in which the object signing system is incorporated. Bamboo is a component framework supporting real-time, networked virtual environments. This design includes dynamic configuration without user interaction such that loadable libraries (modules) are downloaded via HTTP over the Internet and subsequently loaded into memory. Bamboo is designed as a cross-platform tool for the open-source community. [Ref. 5]

5. Implementing Object Signing in Bamboo - Issues

In real-time networks, latency is a pertinent issue. To maintain efficiency in Bamboo's dynamic load process, it is imperative not to introduce additional latency during the module content verification and authentication process. Encryption of the module content is not a requirement and avoids latency associated with expensive encryption and decryption algorithms; this does not pose a security issue since the module content is unclassified in nature. However, modules will have an encrypted message digest attached which will require decryption prior to loading in order to verify content integrity and user authentication. Compressing a module is a requirement before sending across any network to reduce transmission delay. Additionally, modifications to Bamboo must be equally portable across various hardware architectures. To maximize user acceptance and to ensure consistent growth, upgrades must be equally cost-effective to implement, preferably to the extent of freeware. Since there is no requirement for users to sign modules in the Bamboo, it is critical to assess ease-of-use so as not to introduce an encumbrance to the user. Introducing a cumbersome and time-consuming system will preclude use and void the importance of the object signing implementation.

6. Existing Research

While numerous fee-based commercial PKI systems exist today, few object signing systems are readily available. Netscape developed the Object Signing Tool 1.1 for various platforms. This tool creates digital signatures and uses the Sun Microsystem's

Java Archive (JAR) format to associate a signature with files. Digitally signed files using the Object Signing Tool can be verified in Netscape's browser when appropriate certificates are used. Microsoft's Signcode utility is similar to Netscape's Object Signing Tool yet is tailored for the Microsoft browser and is used to sign Microsoft Cabinet Files (CAB). CAB is similar to a ZIP file. The JAR file format is also based on the popular ZIP file format and is used for aggregating many files into one. Although JAR can be used as a general archiving tool, the primary motivation for its development was to allow Java applets and their requisite components to be downloaded to a browser in a single HTTP transaction. The JAR format also supports compression (reducing the download time) and is cross-platform. Additionally, individual entries in a JAR file may be digitally signed to authenticate their origin.

C. THESIS ORGANIZATION

The thesis is organized as follows: Chapter II introduces terminology in PKI systems and explores the requisite background material on a typical PKI system and a typical object signing system. Chapter III discusses the selection criteria for the implemented system and covers tradeoffs in some of the systems available today. Chapter IV discusses the details of how the object signing implementation was achieved. It also covers the pitfalls encountered during the attempted implementation of various other systems before settling on the current system. Chapter V analyzes the results of the security implementation in Bamboo and discusses risks associated with a typical PKI implementation. Chapter VI concludes the thesis and primarily discusses future work and areas of further investigation paralleling the research of this thesis. It reviews some of the attempted implementations of object signing systems and discusses future enhancements, particularly with regard to the failed implementations. It discusses what requirements need to be met before accomplishing a migration to another system providing more robust features.

Two appendices are included as part of the thesis. Appendix A is a User's Guide with detailed installation procedures and illustrates requisite commands to employ Pretty Good Privacy (PGP) v2.6.2 with Bamboo. It also covers procedures such as: how to sign modules and display signatures attached to modules; how to store signed modules on a

server and correspondingly download signed modules from a server; how to store a user's PGP public key on the PGP Public Key Server; and how to extract other users' public keys. The PGP Public Key Server is instrumental as a distribution medium for PGP public keys so that end-users can validate signatures attached to downloaded signed modules. Appendix B details the administration and management of the PGP Public Key Server and is not intended for the end-user.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

Security systems have evolved rapidly in the past 12 months and the industry has entered a consolidation phase with various acquisitions and mergers taking place. Currently, VeriSign has the largest installed user base of customers utilizing their PKI technology and owns Thawte, one of the dominant players in this field. Additionally, they have established strategic relationships with Netscape, Network Associates, RSA, and Security Dynamics. Numerous smaller companies have entered the market in just the past year offering similar PKI security solutions in an effort to take advantage of the rapid explosion in Internet growth. Despite the voluminous number of participants, the basic PKI solution appears to be consistent among the offerings. Emerging technology in the security area of the Internet and networks in general, includes recently available applications of object signing or code signing. To better understand this thesis' implementation of object signing, it is necessary to explain the PKI system in broad terms to comprehend the foundation upon which object signing has its roots. Much of the terminology and definitions present in the PKI system are common in an object signing system.

A. INTERNET SECURITY ISSUES

Communication across the Internet primarily uses the TCP/IP protocol and its flexibility has led to worldwide acceptance. The information is transmitted in TCP/IP packets via a local server and may encounter numerous routers prior to reaching the destination server for end-user processing. Since the data is being transmitted beyond an organization's firewall, it becomes exposed to third-person parties whom may target the transmitted data in various ways. Some of the more common techniques of targeting data packets are [Ref. 6]:

- **Eavesdropping** – Information remains intact but its privacy is compromised.
- **Tampering** – Information is changed or replaced and sent on to the recipient.
- **Impersonation** – Information passes to a person who poses as the intended recipient and consists of two variations:
 - **Spoofing** – A person can pretend to be someone else.
 - **Misrepresentation** – A person can misrepresent oneself.

- **Repudiation** – A person falsely denies that a transaction occurred or was authorized after the fact.

To address these concerns, the PKI system was developed with the necessary features to counter any attempted violation of the transmitted data.

1. Encryption/Decryption

Encryption and decryption is the process of transforming information into unintelligible data by the sender, and at the recipient's end, the data is transformed back into its original state for end-user assimilation. By using this technique, eavesdropping is rendered useless unless the perpetrator has the ability to decrypt the transmitted information.

2. Digital Signatures

To counter the problem of tampering, digital signatures (also referred to as message digests) were developed; a digital signature relies on the use of a mathematical function called a one-way hash routine. The resultant value from a specific hash routine is unique and of fixed length. Changing one character in the original data and hashing the data will result in a totally different hash value. Additionally, the content of the original data cannot be deduced from the resultant hash, hence the name one-way hash. (Popular hash functions include: Message Digest 5 (MD5), Secure Hash Algorithm (SHA), Tiger, and RIPEMD-160.) At the recipient's end, the received data can be input into an identical hash routine and the resultant hash compared with the transmitted hash. However, this technique alone is not sufficient to prevent a perpetrator from violating the integrity of the data. For instance, one could replace the original message and its hash with another. This would ultimately result in the revised transmitted data having a matching hash at the recipient's end and possibly without the recipient's cognizance of the content integrity violation. To completely remove tampering, the resultant hash value must be encrypted with the sender's private key prior to transmission. The encrypted hash along with other pertinent information such as the hash algorithm is known as the digital signature. Again, if the perpetrator is able to decipher the digital signature, then the resultant data can be altered. This provides added significance to the security of one's private key.

Figure 2.1 below encapsulates the digital signature concepts introduced thus far. It illustrates the transmission of a document with a message digest attached to counter the threat of tampering. The sender's private key is used to encrypt the message digest prior to transmission. To validate the integrity of the data at the recipient's end, the original document is input into the identical hash routine used by the signer and compared with the decrypted hash associated with the document. The signer's public key is used to decrypt the message digest; the public key is part of the digital certificate and is extracted for decryption. Finally, the new hash is compared against the originally transmitted hash. If they do not match, the data may have been tampered with since it was signed, or the signature may have been created with a private key that does not correspond to the public key presented by the signer. If the two hashes match, the data has not changed since it was signed and the recipient can be certain that the public key in the signer's certificate corresponds to the signer's private key.

To confirm the identity (authentication) of the signer, however, also involves confirming the validity of the digital signature of the Certificate Authority (CA) issuing the signer's certificate. With the advent of numerous organizations instituting their own private CA servers, there is an inherent risk associated with the acceptance of a certificate from any particular CA. Therefore, the recipient determines the level of risk it is willing to accept when processing certificates affiliated with a specific issuing CA. For instance, it may not be prudent to accept a document with a corresponding digital signature if the certificate was issued by a CA called "Hackers Anonymous". An end-user utilizing today's most common Internet browsers can control which CA issuing agencies to accept certificates from simply by specifying them in the browser itself.

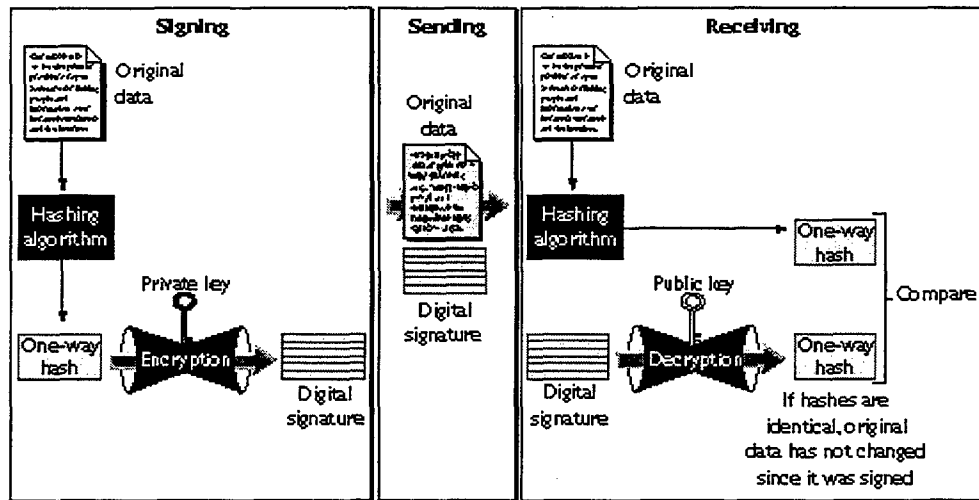


Figure 2.1 Using a digital signature to validate data integrity. From Ref. [6].

The significance of a digital signature is comparable to that of a handwritten signature. Once you have signed data, it is difficult to deny doing so later--assuming that the private key has not been compromised or out of the owner's control. This resultant characteristic of digital signatures is described as non-repudiation. In some situations, a digital signature may be as legally binding as a handwritten signature. Therefore, signers should take great care to ensure that they can stand behind any data they sign. [Ref. 6]

3. Trust in Cyberspace

The anonymity of users poses a threat to the security of information transmitted across networks. In the physical world, trust is built through a complex web of social, legal and business interactions that, in some cases, have taken generations to mature. We use common instruments to establish trust every day such as drivers' licenses, employee badges, credit cards, business licenses, etc. We understand that these instruments are backed by organizations whose diligence, practices, policies, and reputations provide a readily accepted framework of trust for transactions and communications. These means of developing trust cannot be used in a digital world. Additionally, these means of developing trust in the physical world are not failsafe just as they are not in the digital world. [Ref. 7] We investigate a general PKI system to further explore and nurture our degree of trust in cyberspace. In a later chapter, we will investigate some of the risks associated with a typical PKI system and object signing system.

a. Fundamental Goals of a Security System

Electronic security measures are aimed at achieving five fundamental goals [Ref. 8]:

- Ensuring that only authorized individuals have access to information.
- Preventing unauthorized creation, alteration, or destruction of data.
- Ensuring that legitimate users are not denied access to information.
- Ensuring that resources are used in legitimate ways.
- Ease of use so as not to unduly restrict the ability of individuals to go about their daily business.

b. Security Controls

The area of security is an extremely broad field with numerous texts and research outlining various security policies and implementations to achieve a certain level of assurance. Two simple definitions provided to achieve the fundamental security goals listed above are: *communications security*- the protection of information while it is being transferred from one system to another, and *computer security*- the protection of information within a computer system. These broadly defined protective measures can be achieved by implementing established controls such as non-technical and technical security controls. Non-technical security controls include physical security, procedural security, and personnel security. Physical security is concerned with accessibility to the computer security infrastructure to include security protection of the building and room in which the hardware/software is stored; procedural security includes the organization's procedures and policies governing information security; personnel security involves background checks and training. Technical security controls include algorithm strength, operating system strength, auditing organizational security procedures and security breaches, and for the DoD, formal evaluation of components against criteria done by commercial labs in affiliation with joint National Security Agency and National Institute of Standards and Technology (NIST) National Information Assurance Partnership (NIAP). [Ref. 1]

c. Assurance Levels

Assurance levels are typically organization-specific and dictate the desired security policy, procedures, and implementation. One definition which can be applied broadly across various organizations is defined by Rohrbach [Ref. 1]. The levels of assurance are divided into three categories: High, Medium, and Basic. The classifications are grouped by the level of security required based on the type of information to transact. For instance, the High level may be used for command and control/organizational messaging, classified information, and electronic commerce in excess of \$100,000; the Medium level is for individual messaging, legal transactions, and electronic commerce less than \$100,000; all other types of information require only the Basic level of assurance. [Ref. 1]

B. A GENERAL PKI SYSTEM

Two of the most common security precautions in use today are passwords and firewalls. Passwords are designed to prevent unauthorized individuals from directly gaining access to sensitive data stored on computers. Firewalls, by contrast, are designed to provide a perimeter defense mechanism, preventing unauthorized individuals outside the organization from gaining access to sensitive data inside the organization. Despite their important role in network security and widespread adoption, firewalls provide only a partial solution. Any time that data is sent between a server and organizations outside the firewall, the data can be intercepted using "sniffers". Hackers do not need to get into your system if you are sending data outside the perimeter of the firewall. PKI provides tools to achieve security objectives for transporting data beyond the firewall. [Ref. 8]

The following paragraphs offer a detailed discussion of basic terminology employed in PKI technology.

1. Digital Certificates

Envelopes and secure couriers have been replaced with sophisticated methods of data encryption designed to ensure a message is read only by an intended recipient. Physical signatures and seals have been replaced with digital signatures to ensure messages came from a particular identity and that the message was not altered in any fashion during transit. Understanding digital certificates is central to understanding PKI

systems. It is a credential issued by a trusted authority that individuals or organizations present to prove their identity (see Certificate Authority below).

In physical transactions, the challenges of identification, authentication, and privacy are solved with physical marks, such as seals or signatures. In electronic transactions, the equivalent of a seal must be coded into the information itself. By checking that the electronic seal is present and has not been broken, the recipient can confirm the identity of the message sender and ensure that the message content was not altered in transit. To create an electronic equivalent of physical security, digital certificates use advanced cryptography. [Ref. 8] A digital certificate employs a matched pair of keys, private and public keys, that uniquely complement each other such that a message encrypted with the private key can be decrypted with the corresponding public key. Additionally, a message encrypted with the public key can be decrypted with the corresponding private key. The public key may be distributed to other users while the private key is for the sole use of the rightful owner. Compromising the private key will render the PKI system for that individual useless as all security objectives may then be easily breached. Public keys may be easily distributed as part of the digital certificate and therefore effortlessly extracted by users whom trust your certificate. An example of encrypting information using digital certificates is illustrated in Figure 2.2.

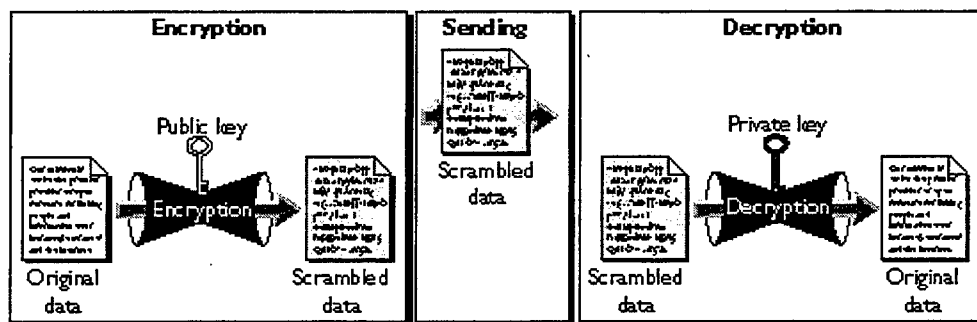


Figure 2.2 Public key encryption/private key decryption. From Ref. [6].

Conversely, you can use your key pair to digitally sign a message with the private key and decrypt the scrambled message with the corresponding public key. A digital certificate is merely a binary file containing pertinent information about the owner such as name, public key, serial number, expiration date, and information about rights and user privileges. Most importantly, a certificate always includes the digital signature of the

issuing CA. The CA's digital signature allows the certificate to function as a "letter of introduction" for users who know and trust the CA but don't know the entity identified by the certificate. Current specifications of digital certificates are available by reviewing the latest X.509 Certificate Standard.

a. Types of Digital Certificates

There are various types of digital certificates on the market today. However, with the recent development of object signing tools, some of the later browsers are incapable of handling the latest certificates such as object-signing certificates. Below are a few of the common certificates available today [Ref. 9]:

- **Client certificates.** Used to identify clients to servers.
- **Server certificates.** Used to identify servers to clients.
- **S/MIME certificates.** Used for signed and encrypted email.
- **Object-signing certificates.** Used to identify signers of Java code, JavaScript scripts, or other signed files.
- **CA certificates.** Used to identify a CA.

b. Private and Public Key Generation

The public and corresponding private keys are created locally by server or client software requesting the certificate. To help ensure that the private key is not compromised, it remains on the local machine, and client software or the system administrator submits the public key along with other information to the issuing CA for the digital certificate. Again, the CA may be an independent company that issues certificates for a fee or an administrator running a Certificate Server within an organization. The CA verifies the identity of the requestor according to the CA's security policies, then issues the certificate.

c. Public Key Algorithms

As illustrated in Figure 2.2 above, an algorithm is necessary to encrypt and decrypt the transmitted data; various public key algorithms are used to accomplish this function. As stated before, a different key is used for encryption and decryption, and the decryption key cannot be derived from the encryption key. Public key methods are important because they can be used to transmit encryption keys or other data securely

even when the parties have no opportunity to agree on a secret key in private. All known methods are quite slow, and they are usually only used to encrypt session keys (randomly generated "normal" keys) that are then used to encrypt the bulk of the data using a symmetric cipher. The following are some of the common algorithms [Ref. 10]:

- **RSA** (Rivest-Shamir-Adelman) is the most commonly used public key algorithm. It can be used both for encryption and for signing. It is generally considered to be secure when sufficiently long keys are used. The security of RSA relies on the difficulty of factoring large integers. Dramatic advances in factoring large integers would make RSA vulnerable. RSA is currently the most important public key algorithm. It is patented in the U.S. (expires Sep. 2000), and free elsewhere.
- **Diffie-Hellman** (DH) is a commonly used public-key algorithm for key exchange. It is generally considered to be secure when sufficiently long keys and proper generators are used. The security of Diffie-Hellman relies on the difficulty of the discrete logarithm problem (which is believed to be computationally equivalent to factoring large integers).
- **DSS** (Digital Signature Standard). A signature-only mechanism endorsed by the U.S. Government. It is intended to provide the capability for the creation and verification of digital signatures and not for use as a general encryption algorithm, nor for key distribution. Its design has not been made public, and many people have found potential problems with it (e.g., embedding hidden data in the signature and revealing your secret key if you ever happen to sign two different messages using the same random number).
- **El Gamal** public key cryptosystem. Based on the discrete logarithm problems in a finite field similar to DH, but has been extended to elliptic curves for greater strength.
- **LUC** is a public key encryption system. It uses Lucas functions instead of exponentiation. Its inventor Peter Smith has since then implemented

four other algorithms with Lucas functions: LUCDIF, a key negotiation method like Diffie-Hellman; LUCELG PK, equivalent to El Gamal public-key encryption; LUCELG DS, equivalent to El Gamal digital signature; and LUCDSA, equivalent to the U.S. Digital Signature Standard.

Further details on public key algorithms is available through most cryptography texts.

2. Registration Authority

An optional component to a PKI architecture is the Registration Authority (RA). The RA may be part of the CA or a separate entity. It is dedicated to user registration and accepting requests for certificates. User registration is the process of collecting user information and verifying a user's identity, which is then used to register a user according to a security policy. This is distinct from the process of creating, signing, and issuing a certificate. An example of an RA in an organization having their own CA may be characterized by a Human Resources department managing the RA function while an Information Technology department manages the CA. A separate RA also makes it harder for any single department to subvert the security system. Systems performing RA and CA functions are often referred to as Registration Servers and Certificate Servers respectively. [Ref. 4]

3. Certificate Authority

The core of the PKI architecture is the Certificate Authority (CA). Digital certificates form the basis of trust and interoperability for a network to communicate securely. As a result, the quality, integrity, and trustworthiness of a PKI system depend on the infrastructure and practices of the CA who issues and manages the certificates. Certificate authorities are entities that validate identities and issue certificates. They can be either independent third parties or organizations running their own certificate-issuing server software. Recently, numerous software packages have been introduced for organizations to host their own Certificate Server instead of outsourcing the responsibility to a third party. (For instance, Netscape Certificate Management System was released in May 1999.) The methods used to validate an identity vary depending on the security

policies of a given CA. Some of the primary duties for a CA and RA include the following:

- Receiving applications and registering identities of end users.
- Validating identities.
- Issuing digital certificates.
- Revoking, renewing, and managing expiration of certificates.
- Maintaining a database of valid certificates and invalid certificates.
- Maintaining security on the CA's private key.
- Ensuring widest dissemination of the CA's digital certificate.
- Establishing trust among members.
- Handling risk management.

In general, before issuing a certificate, the CA must use its published verification procedures for that type of certificate to ensure that an entity requesting a certificate is in fact who it claims to be (background authentication check). If present, an RA would accomplish this function. It is imperative for the CA to maintain rigorous procedural and quality control standards in the authentication of publishers and in the creation, issuance, and maintenance of certificates. When a CA such as VeriSign, Entrust, etc., issues digital certificates, it verifies the owner is not claiming a false identity and is officially providing generally recognized proof of a person's identity. By vouching for an individual's or organization's identity, it is putting its "name" behind the individual's or organization's right to use their own name. The verification process may encompass days prior to the issuance of a certificate from a third party CA; one of the benefits of having the CA in-house is that the time to verify and issue a certificate may be measured in minutes. This verification and issuance process is analogous to the U.S. Government issuing a passport to an individual verifying one's own identity.

a. Issuing Digital Certificates

Upon completion of validating a requester's identity, the CA issues the certificate binding the requester to the submitted public key. This certificate is valid for a designated time period and becomes invalid upon expiration. Therefore, certificate renewal procedures are critical to keep an organization's certificates in the mainstream to

be used by network resources. Typically, the same certificate can be used once the organization pays the required fee to the CA for reissue (if using an outside party as CA). Issued certificates are also stored in a CA repository for future access.

b. Revoking Digital Certificates

Occasionally, it may be necessary to revoke a certificate prior to its expiration date. Occasions for this may be due to the termination of an employee or compromise of a private key. When a certificate is revoked, it is typically placed on a Certificate Revocation List (CRL). If an organization receives a transaction with a digital certificate, it can verify the validity of the digital certificate in various ways. One method is to query the issuing CA each time to verify the certificate is currently valid. This is called real-time status checking and may be computationally expensive. Another technique is for organizations to retrieve the CRL from the CA via a directory service such as the Lightweight Directory Access Protocol (LDAP). LDAP is a directory accessing system for an organization to assist in the management of certificates. System administrators can store much of the information required to manage certificates in an LDAP-compliant directory thereby replacing the real-time status checking at the CA with a rapid directory search at the local level. An LDAP client connects to an LDAP server using a TCP/IP connection and can perform authorized functions such as: Read, Add, Modify, Delete, and Search on specific entries. LDAPs at a minimum typically contain the CRL and serve as a repository for issued certificates. In this manner, a transaction's digital certificate can be verified against the revocation list and processing halted if invalid. Another approach at the organizational level is for the system administrator to maintain a database of valid certificates such that if the certificate attached to a transaction is not in the database, then the transaction would not be processed. Of course, if the organization processes a large quantity of certificates, then the database of valid certificates can become unmanageable in size resulting in increased access times.

c. Storing and Retrieving Digital Certificates and CRLs

The most common method of storing and retrieving certificates and CRLs is via the LDAP directory service. Other retrieval techniques include: e-mail, FTP, and HTTP.

4. Obtaining a Digital Certificate

The following illustrates a simplified procedure to apply and receive a digital certificate from a CA using a hypothetical company called "MyCo". There will be minor differences between respective CAs, however, the principle is the same. Refer to Figure 2.3 below. MyCo provides evidence of its identity, generates a public/private key pair, and sends the evidence and the public key to the CA. The CA then uses the evidence provided by MyCo to verify the identity of the entity requesting a certificate. This may be a superfluous procedure for a free certificate or an extensive verification process requiring numerous days. It is important to request a certificate with the appropriate level of identity verification for the intended use.

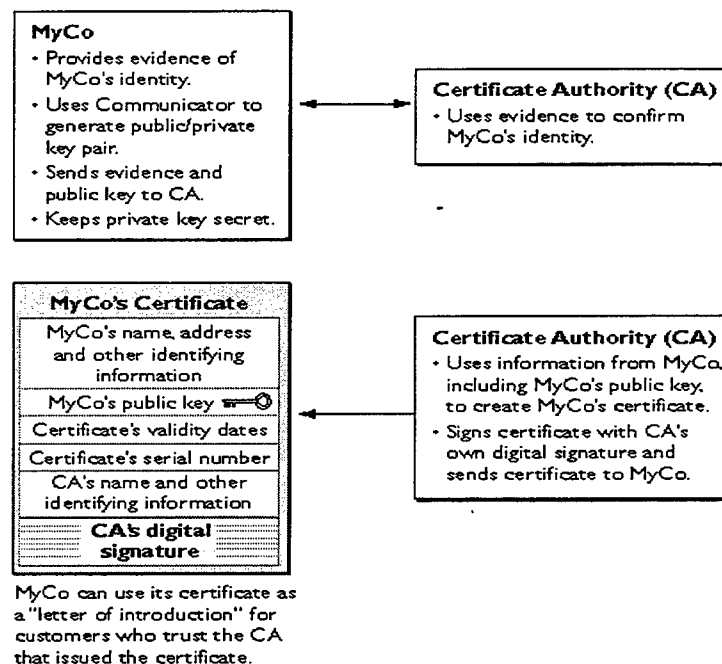


Figure 2.3 Obtaining a Digital Certificate. From Ref. [6].

After the CA has confirmed MyCo's identity, the CA creates a certificate that includes: MyCo's name; its public key; other information, such as the dates during which the certificate is valid and the certificate's serial number; and most importantly, the CA's digital signature. The CA's digital signature allows MyCo to use the certificate as a "letter of introduction" for customers or others who may not be familiar with MyCo but who do know and trust the CA. The CA's signature is obtained by encrypting a one-way hash of MyCo's certificate with the CA's private key. [Ref. 6]

5. Key Backup and Recovery

In the event that a user can no longer access one's public key, there needs to be a measure to recover such a missing key. The first and foremost reason that this may occur is human forgetfulness. Forgetting one's password can prove disastrous when the system no longer allows access to your very own public key. Another reason for such a backup and recovery requirement is that the devices storing the keys may become corrupted. Without key recovery, encrypted data can be lost forever. It is incumbent upon the administrator to ensure the appropriate backup and recovery application is available and executed periodically to ensure ready access to all keys. Again, since the private key strictly belongs to the user, this key should not be backed up, otherwise non-repudiation cannot be guaranteed. [Ref. 3]

6. Typical PKI Security Example

Using the background introduced above, the ensuing material illustrates the major concepts of an arbitrary PKI implementation. In Figure 2.4 below, a user generates his public-private key pair and sends his public key to the CA to obtain a digital certificate. For simplification, assume the RA is part of the CA. The CA verifies the identity of the user and issues a digital certificate and places a copy in the directory for others to access.

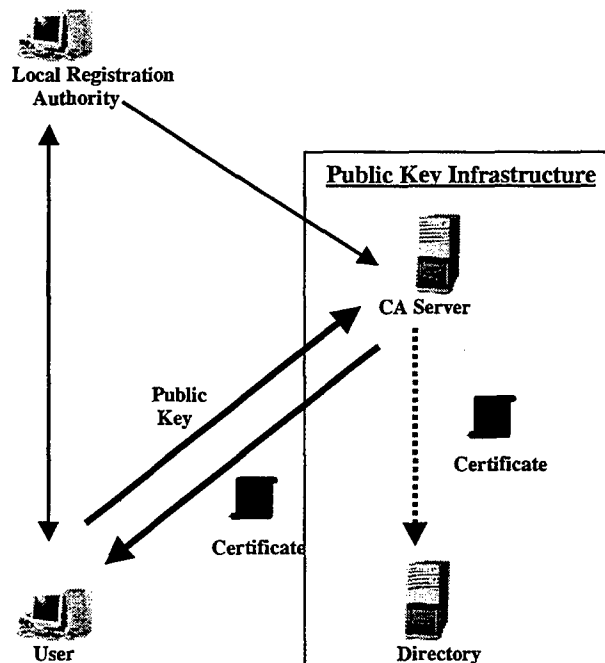


Figure 2.4 A general PKI system.

To continue with the example, assume the user desires to send an encrypted e-mail to another user. As illustrated in Figure 2.5, a user can obtain the recipient's digital certificate containing the recipient's public key. By encrypting the e-mail using the recipient's public key, only the individual holding that recipient's private key can decrypt the e-mail. Given that there has been no compromise of the private key, only the recipient would be able to decrypt and view the e-mail.

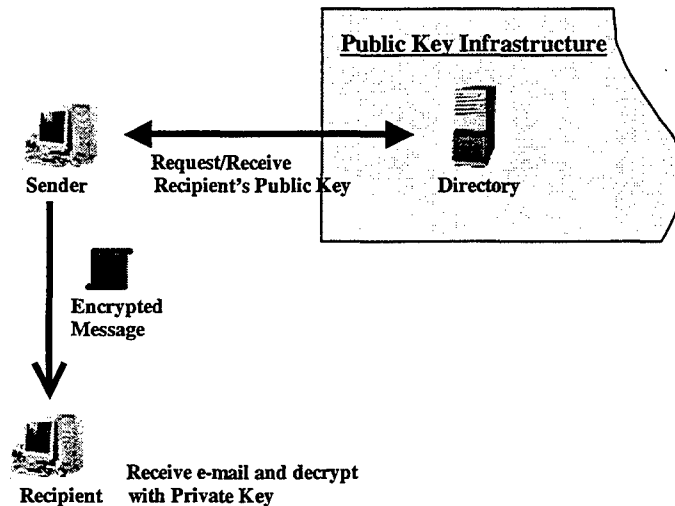


Figure 2.5 Encrypting e-mail.

C. A GENERAL OBJECT SIGNING SYSTEM

The number of Internet users is increasing exponentially. Most users are fluent in acquiring known software from trusted manufacturers through trusted sources such as catalogs or retail stores, but they have little or no experience downloading software from the World Wide Web (WWW) or gauging what effect that software could have on their systems. Organizations are deploying intranets and extranets, tying together employees, customers, suppliers, and service providers who use very different kinds of software on very different platforms. For intranets and extranets, as for the Internet, reliable cross-platform authentication, distribution, and access control for software are becoming more important every day.

Unfortunately, potential security problems arise when data is distributed electronically. This data may be object code, source code, or any other form of data. To address these problems, object signing technology was developed as a means of establishing trust for downloaded software. It allows users to receive downloaded files

with a high confidence level that the content is identical to the original content on the server. Object signing uses standard techniques of public-key cryptography to let users get reliable information about code they download in much the same way they can get reliable information about shrink-wrapped software. It lets end users identify who published software and verify that the software has not been tampered with or altered since the time it was uploaded. Object signing allows developers to digitally sign any type of file that they may wish to distribute over the Internet. Additionally, browsers provided by Netscape and Microsoft have the built-in capability to handle signed objects and process them according to the user's predefined policy. Within the browser, it can also inform users if Java applets are requesting special privileges, such as writing to the hard disk. By presenting this information to users, users can then make informed decisions about downloaded software; for example, whether to allow a signed Java applet to access specific system resources [Ref. 11]. To use signed objects with either of the browsers, you must have an object signing certificate. These certificates can be purchased from third party sources such as VeriSign, or if the organization is running their own certificate server (acting as their own CA), then they would inherently have the capability to issue these certificates.

1. Netscape Object Signing

The Netscape Object Signing system is analogous to the Microsoft Authenticode system despite their incompatibility. Most object signing systems can be characterized with similar traits with modifications to reflect that organization's specific desires. Understanding the primary concepts of this popular object signing system will enable the reader to comprehend the hybrid PGP implementation incorporated into the Bamboo software. Much of the information pertaining to this system was extracted from Ref. [6].

Communicator 4.0 was the first Netscape browser with object signing functionality built in. It was incorporated to help users and network administrators implement decisions about software distributed over intranets or the Internet—for example, whether to allow Java applets signed by a given entity to access specific system resources on specific users' machines.

a. Introduction to Netscape Object Signing

Objects signed by the Netscape Object Signing technology may be any kind of file type. Netscape provides tools that can associate a digital signature with any kind of software object. When a user signs an object utilizing one's private key, the generated digital signature is used in future processing to confirm the identity of the signer and detect tampering that may have occurred since signing (content integrity). The ability to associate a digital signature with a particular entity allows users and network administrators to decide which sources of software they want to trust and to identify software signed by those entities.

b. Features of Netscape Object Signing

By combining the identity and content integrity verification techniques with Java, Netscape has also created a way to control potentially dangerous access by software to local system resources outside of the Java sandbox. The user or network administrator determines what kind of access should be granted or denied for what signers, and Communicator keeps track of the details. [Ref. 6] This privileged access approach may not be prevalent in other object signing schemes. By granting the signed software the ability to perform its required operations without the hindrance of restrictions promulgated by the Java-specific sandbox, it reduces the chances of accidental or malicious damage to the user's system either accidentally by the user or by granting the signed software full-blown access. Another unique feature implemented in Netscape Object Signing technology based on the access privileges, is the capability for the browser to download and execute a signed file while simultaneously granting the appropriate privileges such that the browser can install required plug-ins or other software updates.

c. Archive Files

Netscape Object Signing makes use of the cross-platform Java Archive (JAR) format. JAR archive files, which are compatible with standard cross-platform ZIP files, provide a way to associate digital signatures with specific files in a directory without making any changes to those files. Because the JAR format does not require a digital signature to be stored physically inside the file with which it is associated, JAR

archives can be used to package and sign any kind of files, including file types that have not yet been invented. Users can download signed objects on any computer that can run Netscape Communicator.

d. How Object Signing uses Public Key Cryptography

As depicted in Figure 2.1 above, Netscape Object Signing utilizes the private-public key pair. Private-key encryption is very useful since it allows one to use their private key to sign data creating a digital signature. Communicator (with the aid of the public key) can then confirm that the message was signed with the appropriate private key and that it has not been tampered with since being signed. [Ref. 6]

The key pair used for object signing is identical to the key pair used in PKI. Prior to applying for an object signing certificate from a CA, Communicator generates a public key and the corresponding private key. The certificate issued by the CA binds the public key to the name of the requesting entity. Communicator recognizes several kinds of certificates, including those used to identify e-mail recipients and websites. When you receive a signing certificate for your own use, it is automatically installed in your copy of Communicator. Communicator also supports the public-key cryptography standard known as PKCS #12, which governs key portability. This means, for example, that you can move signing certificates from one computer to another on credit-card-sized devices called smartcards. Signing tools such as the JAR Packager allow you to choose which signing certificate you want to use at the time of signing. [Ref. 6]

To confirm a signer's identity, Communicator relies in part on its list of accepted CAs. A CA can be a publicly recognized independent company such as VeriSign, or it can be an individual or department recognized only within a corporation's intranet or extranet. The user can add CAs to Communicator's list of CAs and, if necessary, delete from the list any CAs that the user decides not to trust for the purpose of validating a digital signature. If a signer's signing certificate cannot be traced back to one of the CAs on Communicator's list, that signer's digital signature cannot be validated.

e. Digital Signatures in Netscape Object Signing

Netscape Object Signing technology utilizes a signing tool to sign an object such as JAR Packager, Page Sign, or Netscape's recently released Sign Tool. This

object is first input into a one-way hash routine and results in a fixed length hash. The resultant hash is encrypted with the private key. The encrypted hash and related information are collectively known as the signer's digital signature. Again, refer to Figure 2.1 above for an illustration. Finally, upon receiving the signed object, Communicator hashes the object and compares the new hash against the original hash to verify content integrity of the object.

An individual file can potentially be signed with multiple digital signatures. For example, a vendor might sign the files that constitute a software product to prove that the files are indeed from a particular company. A network administrator might sign the same files with an additional digital signature based on a company-generated certificate to indicate that the product is approved for use within the company. Figure 2.6 illustrates a file called "MyCoFile.class" which has been signed by MyCo's private key using a signing tool. The signing tool will prepend the digital signature for later content integrity verification and the MyCo digital certificate. Once the object is signed, it may be placed on a web site for download.

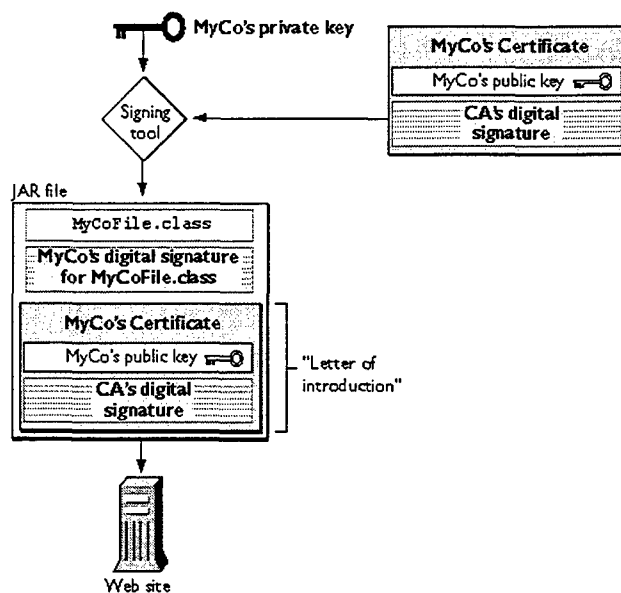


Figure 2.6 Signing an Object. From Ref. [6].

Once the signed object is stored on a web site, it becomes available for download by various users. Figure 2.7 illustrates a signed jar file downloaded from a website. A few assumptions made for this figure include: assumes certificate for the CA that issued the signing certificate is in Communicator's list of accepted CAs, the JAR file

contains only one signed file, and that the signer is not known to the user.

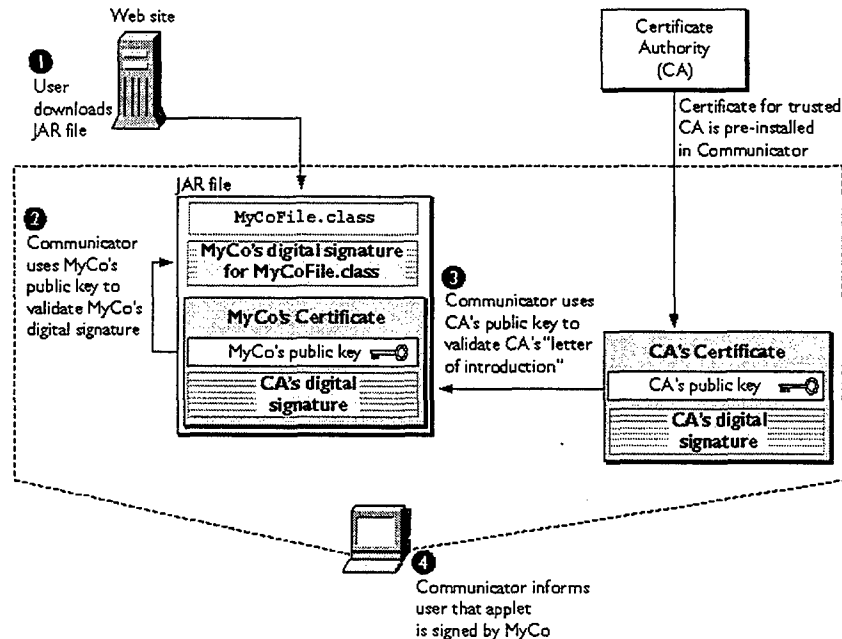


Figure 2.7 Downloading a signed object. From Ref. [6].

The following steps are shown in Figure 2.7 [Ref. 6]:

1. The user downloads the JAR file.
2. Communicator locates the file `MyCoFile.class` and the corresponding digital signature in the JAR file. Communicator then uses MyCo's public key, which it gets from MyCo's "letter of introduction" (MyCo's signing certificate) in the same JAR file, to validate the integrity of `MyCoFile.class`. At this point, Communicator has established that `MyCoFile.class` has not been tampered with since it was signed, and that the public key in MyCo's certificate corresponds to the private key used to sign the file. However, MyCo's identity has not yet been validated; the certificate might have been created by someone attempting to impersonate MyCo. To be sure that the signing certificate really was issued by the specified CA to the real MyCo, Communicator must also perform step 3.
3. Using the public key from the specified CA's certificate in its list of CAs, Communicator validates the CA's digital signature for MyCo's certificate.

4. If steps 2 and 3 are successful, Communicator informs the user that the applet is signed by MyCo. If MyCo's public key cannot validate MyCo's signature for the signed file, or if the CA's public key cannot validate the CA's signature, Communicator informs the user that the signature is invalid.

When several levels of CAs are involved, Communicator can use the same technique to check the validity of not only the CA's public key, but also the public key of the CA that issued the first CA's certificate, and so on. This process of checking the certificate chain continues until Communicator reaches a CA that is included in its list of accepted CAs. If Communicator cannot successfully traverse the certificate chain and identify a CA in its list, it won't accept the original digital signature. [Ref. 6]

2. Other Object Signing Systems

Netscape Object Signing was used to illustrate a general object signing system simply because of the explicit documentation available and familiarity based upon an independent project incorporating the technology. Even though other systems were reviewed such as Microsoft Authenticode, PGP, and various PKI packages capable of object signing, the necessary detail was not easily extractable for illustration purposes. Additionally, it establishes a baseline from which to compare the selected system.

D. SUMMARY

Understanding the concepts introduced thus far will enable the reader to better comprehend the requirements, selection criteria, and implementation specifics for the object signing system in Bamboo. It is also important to recognize that Public Key technology is emerging and has not emerged. The products are maturing and currently have limited integration between systems. Additionally, despite the rapid growth in Public Key implementations, especially in commercial organizations, there is limited operational experience in the field. Ideally in the future, vendors of Public Key technology will deliver integrated and interoperable systems.

THIS PAGE INTENTIONALLY LEFT BLANK

III. SELECTION

In order to select an object signing system, one must define the functionality required of such a system and identify the criteria by which it is bounded (constrained). The functionality can be formalized via a software requirements specification and the implementation specifics via a design specification. These will enable the developer to determine the user's needs and produce a suitable system to meet these needs. Later, additional criteria may be attached to the system restricting the overall design and implementation of such a system, and tradeoffs may need to be made. For instance, resources available to expend toward such an implementation, whether they are one-time or recurring allocations, need to be assessed prior to implementation. Some obvious examples include financial expenditures, personnel requirements to operate and maintain the system, and the space required for additional personnel or equipment. This procedure is not as trivial as may appear.

When researching available systems via direct vendor communication or through web research, the tendency is to acquire the notion that the implementation is relatively benign with respect to initial installation costs and projected maintenance. "Turnkey solution", "low or no maintenance", and "cost savings" are phrases that appear in product literature. However, since the ultimate goal is to identify a suitable system, one must discard the banter, locate acceptable systems that best achieve the specified requirements, install the selected system, and ensure it satisfactorily meets the desired functionality. If unsuccessful, then recurse on the above procedure to identify another system, or alternatively loosen constraints.

The following sections detail the functional requirements identified prior to system selection and some of the research on the multitude of systems available.

A. SOFTWARE REQUIREMENTS SPECIFICATION

The Software Requirements Specification was submitted in June 1999 and describes the requirements set forth for a module signing application in Bamboo. It consisted of two parts: *requirements analysis* to understand the client's (user's) needs and constraints, and *requirements specification* to document the requirements. These activities are logically distinct but occur together since one must identify the user's needs

and constraints in order to properly document them. This thesis summarizes some of the pertinent information contained in the original document and provides the reader with the original problem statement.

1. Original Stated Purpose

The primary goal of this application is to develop a process where users can download and upload software modules using Bamboo and receive a degree of content integrity and user authentication. This application is required to counter any malevolent threat against software distributed while using Bamboo.

2. Requirements

Two of the primary functional requirements are:

- content integrity
- user authentication

The primary non-functional requirements (constraints) are:

- maintaining performance (speed and simplicity)
- portability
- cost

Since module signing in Bamboo is an optional feature, a failure to maintain any of the following will surely preclude the endorsement by end-users: efficiency, low or no additional cost, minimal training, simple installation and use. Additional functional and non-functional requirements are highlighted below.

3. Rationale

Bamboo allows modules to be dynamically loaded from a server. Additionally, clients can upload their respective modules to a server for mutual access by other clients (dependent upon granted permissions). However, there is currently no degree of assurance that a downloaded module is identical in content to the module initially stored on the server. The module may have been tampered with during transit to and from the server or it may have been altered due to noise during the transmission sequence. Additionally, establishing confidence that the client uploading a module is actually the person claimed to be is of relevant importance. By using signatures, one can establish authorship of a respective module.

A full-fledged PKI system would more than solve the implementation issues desired for Bamboo. The drawbacks to such an implementation are primarily due to the nature of Bamboo; it is freeware for public use and dissemination. Bamboo loads modules in an expedient manner; these significant traits would be lost with time lags associated with full encryption and decryption inherent in a PKI system. Additionally, some clients may utilize Bamboo with no inclination to sign modules. The primary drawback to a full PKI system is the expense associated with installation and maintenance.

4. Scenarios

To further delineate the required functionality of the module signing application, the following scenarios have been formulated. The scenarios are divided into "usual" and "counter-malicious" scenarios; the "usual" scenarios are designed to illustrate normal procedures performed by a user and the "counter-malicious" illustrate implemented lock-outs to prevent a detrimental or undesired action. These scenarios present the principal user requirements to the reader.

a. Usual Scenarios

- User uploads created module to server after signing the module then subsequently downloads the same module from the server. This represents the "usual" mode of operation when a developer will consistently download his own module, make modifications, test changes, sign it, then upload it again. See Figure 3.1.

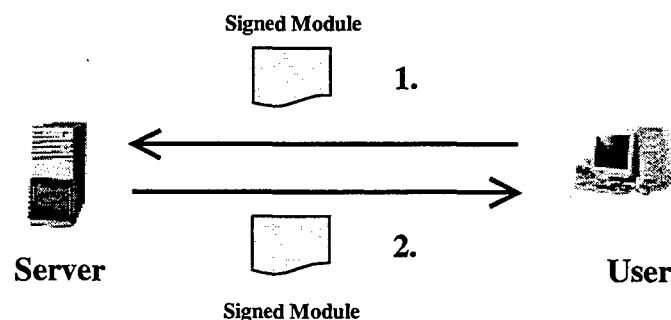


Figure 3.1 Normal Download of a Signed Module.

- User is running Bamboo when a specific module is required for dynamic load. Bamboo dynamically loads module and checks for

content integrity similar to Figure 3.1 above. Upon verifying the content integrity, processing proceeds. See Figure 3.2.

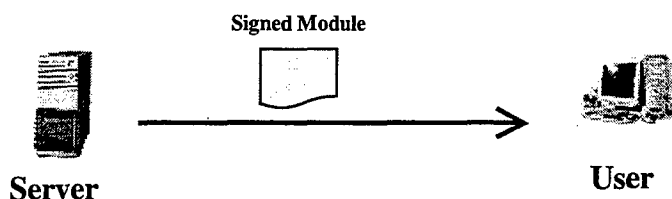


Figure 3.2 Dynamic Download of a Signed Module.

- User A physically gives his signed module to User B. User B decides he “trusts” the module and would like to attach his name along with User A’s. User B “signs” the module and returns the module to User A. User A uploads the module to the server with both names attached. See Figure 3.3.

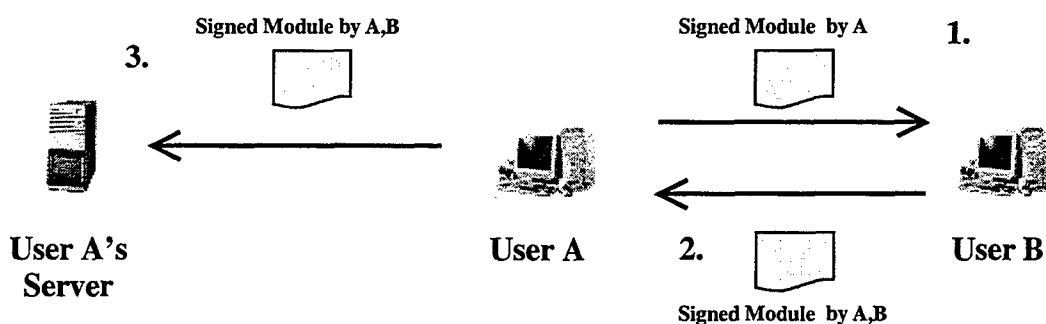


Figure 3.3 Multiple Signatures per Signed Module.

- User downloads an “unsigned” module from the server (considered “unsafe”). Upon receipt of the “unsigned” module, the user will be warned prior to unzipping and unarchiving the file that the user assumes any risks associated with using an “unsafe” module. User is provided the option of deleting the file prior to these operations. See Figure 3.4.

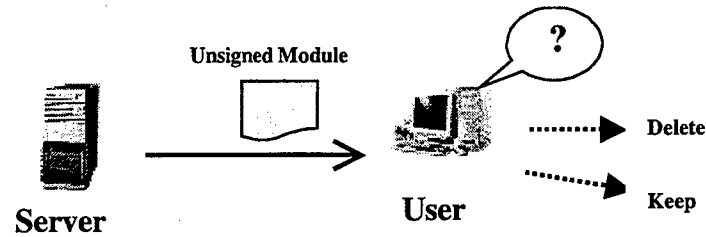


Figure 3.4 Download of an Unsigned Module.

b. Counter-malicious Scenarios

- User downloads a module from the server and the module's hash does not match the prepended hash within the module. Module is considered "unsafe" and is deleted immediately. This module may have been tampered with in transit from the server to client and has content integrity problems. Recommend reattempt download. See Figure 3.5.

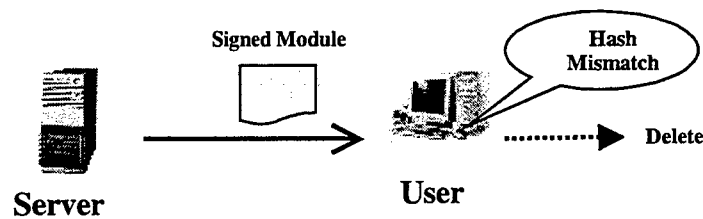


Figure 3.5 Mismatch on Hash Value; Content Integrity Problem; Module Deleted.

- User downloads a signed module but does not possess the signer's public key for digital signature verification. The application will query the user to determine if the module shall be unzipped and unarchived prior to further processing, or if the user desires to delete the module. See Figure 3.6.

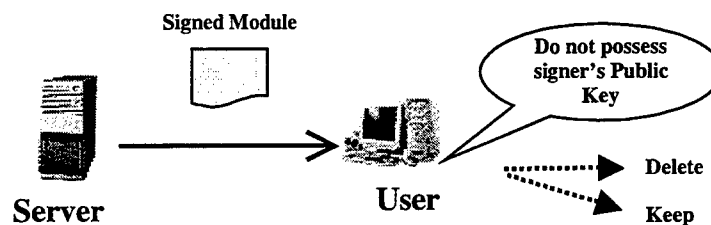


Figure 3.6 User does not Possess Signer's Public Key for Digital Signature Verification.

- User uploads a module to the server. Enroute to the server, the module is tampered with (modified). Upon downloading the same module, a user receives an “unsafe module” error message. Since the module has been tampered with, the digital signature in the original module will not match the new signature since certain values have changed. See Figure 3.7.

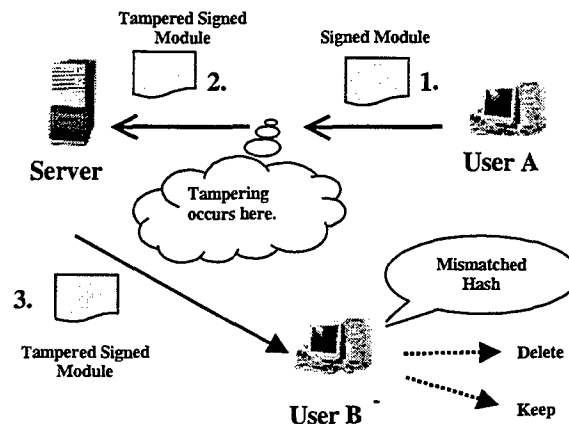


Figure 3.7 Modified Module’s Digital Signature will not match new Hash Value.

5. Requirements Specification

The requirements specification provides the necessary detail to assist the developer in implementing the desired functionality specified above. A few functional and non-functional requirements were previously mentioned and some general concepts pertaining to Bamboo were introduced in Chapter I. Since object signing primarily involves file content verification and manipulation, it is also necessary to define the concepts of *module* and *file type* in Bamboo.

A module can be just about any sub-directory in the Bamboo file system. All the code and data files pertaining to this module reside inside this sub-directory. The current Bamboo “archive” command creates a file with a ‘.tar.gz’ extension after archiving and zipping a module. It is analogous to the JAR file format.

Associated with a module is a file type. The file types in Bamboo are user-specified as *data*, *object*, or *source* during the archive operation. Ideally, the software

should be able to inspect the module contents to affirm the specified type based on specific keyword searches.

Modules loaded from a server via Bamboo consist of either "signed" or "unsigned" executable modules, "unsigned" data modules, and "signed" or "unsigned" source code modules. An "unsigned" executable or source code module is uploaded by a client choosing not to "sign" the module. Any "unsigned" module is deemed "unsafe", however, data modules should remain "unsigned". Bamboo modules are stored with the following form in the user-specified '*bamboo/cache*' directory of the end-user's computer: "*file_name.platform_type.bar*". For instance, if a user on an SGI Irix system archives a module called "*myModule*", the resultant signed archive module would be saved in the '*bamboo/cache*' directory with the name: "*myModule.sgi.bar*". The '*.bar*' signifies Bamboo ARrchive module.

a. Functional Requirement Specifications

- **Purpose:** Content integrity - ensuring a module is identical in content at the server and the client.

Input: Bamboo module directory path.

Process: Presently, the Bamboo "archive" command will tar a directory and gzip a file ('*.tar.gz*' extension). During the archive operation, hashing the '*.tar.gz*' file and prepending the encrypted hash (digital signature) onto the file provides a means for verifying data integrity. When the module is downloaded, its unique hash value will be compared with the prepended digital signature to ensure a match. If there is a mismatch, then the module is different from the time the "archive" command was executed prior to upload.

Output: '*.bar*' file with attached digital signature.

- **Purpose:** User authentication - signing of modules for authorship. This is directly related to previous functional requirement and requires the hash to be computed then signed by the signer's private key resulting in the digital signature.

Input: Bamboo module directory path or '*.bar*' file.

Process: Provide means for user to “sign” module and attach one’s name to the module. The name is also included as part of the file so that it can be used as an index into the recipient’s set of keys or certificates to identify which public key to extract for decryption. Process entails using user’s private key to encrypt prepended hash value on ‘.bar’ file (do not encrypt the file contents).

Output: ‘.bar’ file with digital signature of signer and name attached.

- **Purpose:** Allow multiple signers of a single module.

Input: ‘.bar’ file with zero or more names/signatures attached.

Process: Provide means within Bamboo for user to add one’s name to a module. The module may have names already attached to the module.

Output: ‘.bar’ file with additional name attached.

- **Purpose:** Determine type of module downloaded or uploaded: “signed” or “unsigned” executable, “signed” or “unsigned” source code module, or data module (“unsigned”).

Input: ‘.bar’ file.

Process: Provide means to accept user-specified file type and to verify via content inspection.

Output: Upon file type determination, Bamboo will be able to process module contents as required.

b. Non-functional Requirement Specifications (Constraints)

- **Usability requirements** – Object signing feature should not deter simplicity and usability of Bamboo and should be almost invisible for those not utilizing the security application.
- **Performance requirements** - Performance should not be hampered since only the hash value is encrypted and decrypted and not the module contents.
- **Reliability requirements** - Reliability is crucial to ensure content integrity and authorship of modules. Many recently released software

applications have not passed the “test of time” toward certification or acceptance.

- **Portability requirements** - Must maintain the current portability effort of Bamboo.
- **Overseas usability requirements** – Must be able to either export the system for use outside of the United States (U.S.) or have the ability to legally acquire the system for Bamboo’s International users. Current U.S. export laws deem certain cryptography systems as illegal for export other than to Canada. If the system cannot be exported outside of the U.S., then a compatible system must be available outside of the U.S. for foreign use.
- **Cost requirements** - Must identify initial and recurring annual expenses for system installation, equipment appropriation, and other expenses associated with the user such as certificate requirements. As stated before, an open-source package must maintain the nearly freeware characteristic to end-users with maximum benefit, especially for an optional software enhancement such as module signing.
- **Personnel requirements** – No additional manpower requirements.

With the requirements identified and documented in the Software Requirements Specification, the subsequent step was to research and identify suitable systems.

B. COMMERCIAL SYSTEMS AVAILABLE

Many commercial systems are available on the market today with VeriSign as the leading vendor of PKI technology. A simple filter used to exclude most commercial systems is to review the pricing model for available systems. Figure 3.8 is a chart created from data in Ref. [12] providing a 1998 pricing comparison between Entrust and VeriSign. The data is for the lowest cost scenario employing digital certificates for user authentication in a strictly browser environment. The price covers a five-year time span and is limited to only 5000 certificate holders. A quick glance at the data and one can compute an annual cost exceeding \$100,000, definitely cost-prohibitive for the scope of this thesis.

Five Year Total Cost of Ownership of Basic Certificates for Web Authentication (5000 Users)

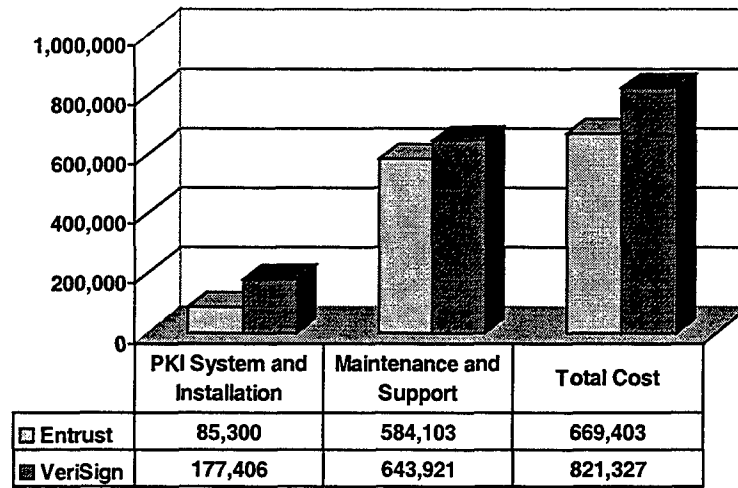


Figure 3.8 Entrust vs. VeriSign Cost Comparison for Five Years. From Ref. [12].

Having quickly eliminated the majority of commercial PKI systems, it became time to explore the opposite extreme, that of freeware.

1. Other Systems Investigated (Chronological Perspective)

Following the suggestion of Professor Brutzman and Kent Watsen at the Naval Postgraduate School, the focus was redirected toward investigating Pretty Good Privacy (PGP), a product from Network Associates. PGP is by far the most widespread software in use for digital document signing. It is freely available all around the world, but you need to download your copy from an appropriate place: outside the U.S. if you are not a U.S. citizen, and from MIT if you are [Ref. 13].

In July 1999, *PGP v6.5.1 Command Line* was released as freeware for non-commercial use. It provides the required functionality such as private/public key encryption, digital signatures, and numerous other features. There is also a freeware Certificate Server available for management of PGP public keys. Since PGP already has the largest user-base and PGP v6.5.1 is compatible with the latest releases, it appeared to be the ideal choice for implementation.

PGP v6.5.1 was subsequently layered on top of Bamboo and the current “unarchive” operation was modified in the `bbModule.c++` file. Additionally, the `archiveApp.c++` file was changed to reflect the new requirements established for the

"archive" command; it used PGP v6.5.1 to provide digital signatures with multiple signatories.

The primary operating systems utilized for testing during this period were Windows NT and SGI Irix. Since PGP v6.5.1 was a recent release with a version for NT/95 and another version compatible with AIX, HP-UX, SUN Solaris for SPARC and Linux Red Hat, it was necessary to contact PGP Technical Support for information pertaining to the anticipated release of other Unix variants. To much chagrin, SGI Irix was not on Network Associates' list of future operating systems to include.

After reverting back to the WWW for further research, one website (<http://wls.wwco.com/security/myca.html>) provided one developer's experience in establishing his own CA to issue certificates using the freeware SSLeay software. One of its links is to the Cryptsoft website (<http://www2.psy.uq.edu.au/~ftp/Crypto>) providing the background and software related to installation. Reviewing the "Frequently Asked Questions" section on the Cryptsoft website provided a wealth of vital information, yet did not provide clear guidance on legal issues in and outside of the U.S. and further recommended consulting a lawyer. It became apparent that this system may have beneficial ramifications in the near future if export laws change. (Export laws changed in January 2000; see Chapter VI.) At the time, cryptography was entangled in a legal mesh and export restrictions were difficult to decipher.

In September, the Clinton administration proposed that "retail" encryption hardware and software of unlimited strength could be exported without a license after a "one-time technical review" and some reporting on whom the products are sold to. "Custom" products would have some restrictions on sales to foreign governments and known terrorist or criminal organizations. Products with key lengths of 64 bits or less would be entirely decontrolled. [Ref. 14] With recent changes in export laws, the SSLeay product is a strong candidate for further research and is discussed in Chapter VI.

In September 1999, the popular GNU open-source community released GPG v1.0, the GNU Privacy Guard (<http://www.gnupg.org>). This security package is similar to PGP and compatible with both PGP 2.x and PGP 5.x. It is designed as freeware even for commercial use. Akin to PGP v6.5.1, a key server is included designed for storage and

distribution of keys. One major benefit of this system is the interoperability between older PGP systems, a characteristic not inherent between all PGP releases. Additionally, it employs the latest security technology at an amenable price, free for all users.

After compiling the source code downloaded from the GNU website, the software was tested on the SGI Irix operating system. The first step with any module signing package is to generate the public and private keys. The key generation procedure failed repeatedly. Further research on the website revealed that this specific "key generation" bug had existed for Irix operating systems in the beta release since the previous Spring and was still present in the GPG v1.0 release. After deliberating whether to devote an indeterminable amount of time to resolve the bug or pursue another avenue, the latter was chosen. The primary reason was that a brand new release usually has other unidentified bugs present that may deviate progress from the predetermined time schedule; focus reverted to another implementation.

C. SELECTING A SYSTEM

Researched systems were grouped into two general categories:

- System requiring annual X.509 certificate purchase
- In-house implementation of CA

1. X.509 Certificate Systems Revisited

In Chapter II, Netscape Object-Signing technology was introduced as a basis for understanding the principles of module signing. With the Netscape tools or even the Microsoft tools, a developer can produce the required code to perform the authentication and content integrity desired in a module signing system. However, one of the drawbacks to utilizing either of these is the need to purchase certificates. Current pricing for an object signing certificate at VeriSign is \$400 annually--a price which may deter many end-users from even considering using the "optional" module signing application.

2. In-house CA System

In-house CA systems can be further divided into two separate categories, X.509 and PGP. The primary differences are discussed in Chapter IV. Briefly with respect to implementation, PGP systems have been in force worldwide for years and have an established user-base with strong open-source technical support. The freeware in-house

X.509 CA systems are relatively new with export restrictions (during selection time) and require extensive technical know-how and overhead to implement and support on a daily basis.

The cost versus benefit tradeoff needs to be analyzed in order to make a determination as to how much security one can afford in time, personnel, and equipment. Again, the intent is to provide a determinable degree of confidence and security with respect to signed modules used by end-users of Bamboo. This subjective matter will be broached further when risks of PKI technology and the implemented system are discussed in Chapter V.

3. Selection

The MIT website ([Ref. 13]) is one of the primary sources for the latest PGP software available as freeware, non-commercial use. PGP v2.6.2 is the "command line" predecessor to the latest release, v6.5.1 (v6.5.2 subsequently released in October 1999). It was the obvious choice to layer on top of Bamboo since it provided the same functionality with just a few restrictions such as key length, a sometimes hastily used quick measure of encryption strength. However, it is still capable of generating keys of 1024 bits. Additionally, the PGP International website ([Ref. 15]) has a compatible version, PGP v2.6.3i.

The PGP Public Key Server software available via MIT has been modified and implemented as the Bamboo PGP Public Key Server. It serves as the central repository for public keys in the Bamboo system and provides a medium by which end-users can access other keys and store their own key. One beneficial feature associated with the free key server software is that an organization can establish their own in-house key server independent of the Bamboo PGP Key Server and obtain the same key management functionality within the confines of the organization.

In summary, there exist a wide array of systems available. The Software Requirements Specification proved invaluable in defining the problem and eliminating certain systems. With the selection made, implementation became the next critical task.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. IMPLEMENTATION

PGP v2.6.2 is available at Ref. [13] for U.S. and Canadian residents. The International version PGP v2.6.3i is available from Ref. [15]. Both versions have been tested under each of the main operating system variants, Windows NT and Unix. Appendix A covers details for installing and compiling the PGP software, how to test the installation, and how to generate and store your PGP keys locally. Additionally, procedures are outlined to archive and sign modules via Bamboo, transfer a module to a server via FTP, download a signed module followed by digital signature verification, and finally, procedures to store and retrieve PGP public keys on the Bamboo PGP Public Key Server (key server). Appendix B is the reference manual for the key server administrator.

The selected version of PGP is a command-line package. Many of the PGP invocations are performed via Bamboo, however, the user requires familiarity with many of the command-line key management commands to perform administrative functions such as *generate*, *disable*, *view*, and *revoke*. Before highlighting the PGP software specifics, the following general discussion is provided as an introduction to PGP v2.6.2. The information was extracted from Refs. [13, 16, 17] and is suggested as recommended reading.

A. PGP KEYS

The two most common certificate types are PGP and X.509. The most salient differences are:

- PGP certificates can be created immediately while X.509 certificates require a CA to issue them and typically with an annual fee.
- PGP certificates can support more than one name for the key owner while the X.509 certificates can only bind a single name to the public key.
- PGP certificates can have more than one digital signature attesting to the certificate's validity while X.509 can only support one, the CA's.

The differences listed above, while providing flexibility, may be construed as PGP's greatest drawbacks. Even though PGP provides rapid certificate generation with multiple names and signatures per certificate, from a security standpoint it is comforting

to have only a single CA attesting to the authenticity of an individual, especially in situations with legal ramifications.

Most secure business applications use an X.509 certificate-based scheme. X.509 has a rigid structure, ASN.1 encoding, and a single issuer (CA). PGP is a flexible “wallet” of signatures over specific attributes using RADIX-64 encoding [Ref. 16]. Digital certificates are just one small component of the bigger PKI picture, but are the fundamental building block that can limit or extend the overall capabilities of a secure infrastructure. [Ref. 18]

PGP keys are kept in individual “key certificates” that include the key owner’s userID (usually name and e-mail address), a timestamp of when the key pair was generated, and the actual key material. Public key certificates contain public key material, while secret (private) key certificates contain secret key material. Each secret key is also encrypted with its own password, in case it is stolen. A “keyring” contains one or more of these key certificates. Users possess two separate keyrings, a public and private keyring, which are used as the principal method of storing and managing keys. These keyrings facilitate the automatic lookup of keys either by keyID or userID. A keyID is an “abbreviation” of the public key (the least significant 64 bits of the large public key). When this keyID is displayed, only the lower 32 bits are shown for further brevity. An individual public key may also be temporarily stored in a separate file to send to an end-user for placement on their public keyring. [Ref. 16]

1. Key Generation/Local Storage

To generate a unique public/private key pair, type “pgp -kg” (key generation) and answer the prompts with regard to key size, userID, and pass phrase. For userIDs, follow the convention: “first-name MI last-name <e-mail address>”. Spaces and punctuation are allowed in the userID. PGP asks for the pass phrase to protect your secret key in case it is compromised. No one can use the secret key without this pass phrase. The pass phrase is like a password, except that it can be a whole phrase or sentence with many words, spaces, punctuation, or anything else. Do not lose the pass phrase—it is unrecoverable if lost and renders the secret key useless. Figure 4.1 below illustrates a sample public/private key generation.

```

C:\pgp262>pgp -kg
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 2000/01/19 15:22 GMT
Pick your RSA key size:
  1) 512 bits- Low commercial grade, fast but less secure
  2) 768 bits- High commercial grade, medium speed, good security
  3) 1024 bits- "Military" grade, slow, highest security
Choose 1, 2, or 3, or enter desired number of bits: 3
Generating an RSA key with a 1024-bit modulus.

You need a user ID for your public key. The desired form for this
user ID is your name, followed by your E-mail address enclosed in
<angle brackets>, if you have an E-mail address.
For example: John Q. Smith <12345.6789@compuserve.com>
Enter a user ID for your public key:
John T. Doe <doe@cs.nps.navy.mil>

You need a pass phrase to protect your RSA secret key.
Your pass phrase can be any sentence or phrase and may have many
words, spaces, punctuation, or any other printable characters.

Enter pass phrase:
Enter same pass phrase again:
Note that key generation is a lengthy process.

We need to generate 768 random bits. This is done by measuring the
time intervals between your keystrokes. Please enter some random text
on your keyboard until you hear the beep:
  0 * -Enough, thank you.
.....**** .....****
Key generation completed.

```

Figure 4.1 Key-pair Generation.

The public/secret key pair is fabricated from large truly random numbers derived mainly from measuring the intervals between keystrokes with a fast timer. The software prompts the user to enter random text to help it accumulate random bits for the keys. The user should provide keystrokes that are reasonably random in timing and different in content since some of the randomness is derived from the unpredictability of the content typed.

The generated key pair will be placed on your public and secret keyrings, `pubring.pgp` and `secring.pgp` respectively. Later, the "pgp -kxa" (key extract ASCII format) command may be executed to extract (copy) your public key to a separate public key file suitable for distribution via e-mail to other users or for storing on the key server.

For instance, Figure 4.2 illustrates the extraction of the public key previously created and stored in an ASCII text file called `doeKey.asc` (note: specifying the `.asc` extension is optional).

```
C:\pgp262>pgp -kxa doe
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 2000/01/19 16:08 GMT

Extracting from key ring: 'pubring.pgp', userid "doe".

Key for user ID: John T. Doe <doe@cs.nps.navy.mil>
1024-bit key, Key ID 67468829, created 2000/01/19

Extract the above key into which file? doeKey

Transport armor file: doeKey.asc

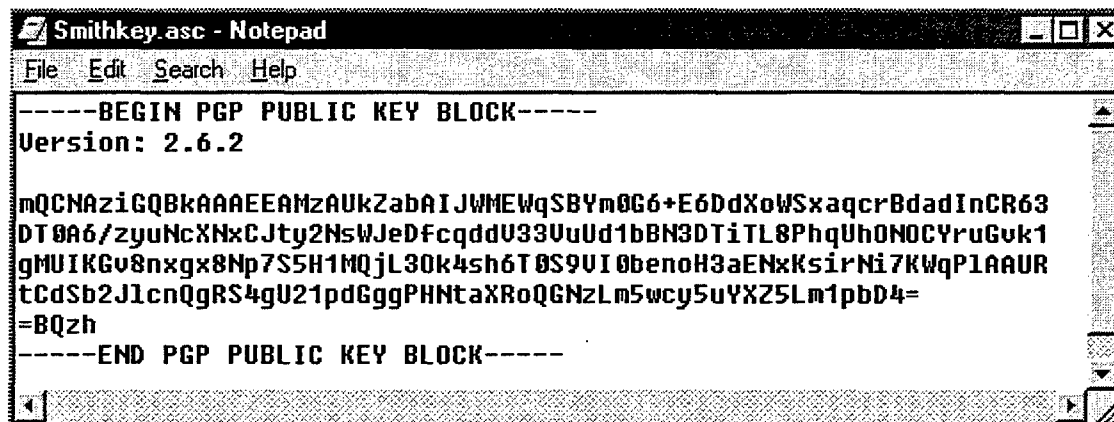
Key extracted to file 'doeKey.asc'.
```

Figure 4.2 Public Key Extraction into ASCII Text File.

Even though each secret key is individually protected via its own pass phrase, never disclose the secret key. Maintain physical control of your secret key, and avoid storing it on a computer accessible via a network.

2. Storing a Public Key on a Keyring

Another person's public key is the fundamental element required to verify a signature on a module signed by that individual. To store another user's public key onto a public keyring, either via e-mail or by downloading it from the key server, obtain the ASCII version of the public key and store it in a text file with a `.asc` extension. For example, using Windows NT, access the key server using Explorer extracting a public key for a fictitious friend Robert Smith. Copy the key material into a Notepad file and save as `"Smithkey.asc"`. Open a DOS window and execute `"pgp -ka Smithkey"` (key add). The PGP software will notify the user that the key has been added to the public keyring. Follow a similar procedure if using e-mail. Duplicate keys will not be added to the keyring. Figures 4.3 and 4.4 illustrate.



```
Smithkey.asc - Notepad
File Edit Search Help
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: 2.6.2

mQCNaziGQBkAAEEAMzAUkZabAIJWMEWqSBYm0G6+E6DdXoWSxaqcrBdadInCR63
DT0A6/zYuNcXNxCJty2NsWJeDfcqddU33UuUd1bBN3DTiTL8PhqUh0NOCYruGuk1
gMUIKGv8nxgx8Np7S5H1MQjL30k4sh6T0S9VI0benoH3aENxKsirNi7KWqP1AAUR
tCdSb2JlcnQgRS4gU21pdGggPHNtaXR0QGNzLm5wcy5uYXZ5Lm1pbD4=
=BQzh
-----END PGP PUBLIC KEY BLOCK-----
```

Figure 4.3 Storing Extracted Public Key Information in an ASCII Text File.

```
C:\pgp262>pgp -ka Smithkey.asc
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 2000/01/19 17:29 GMT

Looking for new keys...
pub 1024/CA5AA3E5 2000/01/19 Robert E. Smith <smith@cs.nps.navy.mil>

Checking signatures...

Keyfile contains:
  1 new key(s)

One or more of the new keys are not fully certified.
Do you want to certify any of these keys yourself (y/N)?
```

Figure 4.4 Saving a Public Key on a Local Keyring.

3. Checking Validity

One of the most threatening attacks on a public key system is for a perpetrator to impersonate someone else. Validity is the confidence that a public key certificate actually belongs to the purported user. Only after a user has assured that a public key belongs to a specified user (binding) can the user sign the public key on the keyring giving his seal of approval. If you want others to know that you gave a particular certificate your stamp of approval, you can export the signed certificate to a certificate server for others to extract and view. In an organization using PGP certificates without a CA such as this implementation, a security policy may have to be defined to perform the function of a CA. One implementation may state that it is the job of the key server administrator to

check the authenticity of all PGP certificates and then sign the good ones. This scenario would ordain the administrator as the final check on certificate binding and the enforcer of certificate trust. This would entail that each end-user maintain the server administrator's public key for digital signature verification. Whatever doctrine is adopted, end-users should always search the key server database and view the latest status of keys.

a. Manually Verifying a Public Key using the Fingerprint

One way to verify a public key is by confirming the PGP certificate's unique fingerprint. The 16-byte fingerprint is a hash of the user's certificate and appears as one of the certificate's properties. Figure 4.5 illustrates the "pgp -kvc" (view fingerprint of public key) command to view Robert Smith's fingerprint. Contacting Robert Smith via telephone and confirming the fingerprint characters ensures the key in fact belongs to Robert Smith.

```
C:\pgp262>pgp -kvc smith
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 2000/01/19 20:24 GMT

Key ring: 'pubring.pgp', looking for user ID "smith".
Type bits/keyID  Date    User ID
pub 1024/CA5AA3E5 2000/01/19 Robert E. Smith
<smith@cs.nps.navy.mil>
Key fingerprint = CA 6D 7A 0F B6 BB 0D B7 24 91 F7 DC B4 E8 F8 0C
```

Figure 4.5 Viewing a Public Key Fingerprint for Verification.

Once verification is complete, then sign the public key with your private key (attaching your approval). By signing the key, other users that trust you will gain the confidence in this same key establishing a "trusted network". Figure 4.6 shows the certification of Robert Smith's public key by signing it with the user's private key.

```

C:\pgp262>pgp -ks smith -u doe
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 2000/01/19 21:12 GMT

Looking for key for user 'smith':

Key for user ID: Robert E. Smith <smith@cs.nps.navy.mil>
1024-bit key, Key ID CA5AA3E5, created 2000/01/19
  Key fingerprint = CA 6D 7A 0F B6 BB 0D B7 24 91 F7 DC B4 E8 F8 0C

```

READ CAREFULLY: Based on your own direct first-hand knowledge, are you absolutely certain that you are prepared to solemnly certify that the above public key actually belongs to the user specified by the above user ID (y/N)? y

You need a pass phrase to unlock your RSA secret key.
Key for user ID "John T. Doe <doe@cs.nps.navy.mil>"

Enter pass phrase: Pass phrase is good. Just a moment....
Key signature certificate added.

Make a determination in your own mind whether this key actually belongs to the person whom you think it belongs to, based on available evidence. If you think it does, then based on your estimate of that person's integrity and competence in key management, answer the following question:

Would you trust "Robert E. Smith <smith@cs.nps.navy.mil>"
to act as an introducer and certify other people's public keys to you?
(1=I don't know. 2=No. 3=Usually. 4=Yes, always.) ? 2

Figure 4.6 Certifying another's Public Key on your Keyring.

In Figure 4.7, the "pgp -kc" (view contents and check certifying signatures of your public keyring) command lists the keys on the user's public keyring and the certification attached to each. In this example under the "Type" heading, there are two keys listed as pub. The first public key listed is Smith and the line below it signifies it has been signed by Doe. Under the "Trust" heading, Smith is listed as Untrusted, and therefore Doe will not trust any keys certified by Smith (see Establishing Trust section below).

```

C:\pgp262>pgp -kc
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 2000/01/19 21:20 GMT

Key ring: 'pubring.pgp'
Type bits/keyID Date User ID
pub 1024/CA5AA3E5 2000/01/19 Robert E. Smith <smith@cs.nps.navy.mil>
sig! 67468829 2000/01/19 John T. Doe <doe@cs.nps.navy.mil>
pub 1024/67468829 2000/01/19 John T. Doe <doe@cs.nps.navy.mil>

KeyID Trust Validity User ID
CA5AA3E5 untrusted complete Robert E. Smith <smith@cs.nps.navy.mil>
c ultimate John T. Doe <doe@cs.nps.navy.mil>
* 67468829 ultimate complete John T. Doe <doe@cs.nps.navy.mil>

```

Figure 4.7 Viewing Certifications of Public Keys.

This verification and certification procedure works if one recognizes the key owner's voice. However, if the key owner is not known or the voice is not recognized, then another method must be pursued. One other way to establish validity of someone's certificate is to trust that a third individual has gone through the process of validating it, whether it be a CA or a third party that you trust. Another important step in checking certificate validity is to ensure that it has not been revoked.

4. Establishing Trust

You validate certificates. You trust people. More specifically, you trust people to validate other people's certificates. Typically, unless the owner hands you the certificate, you have to go by someone else's word that it is valid. [Ref. 13] In most situations, a CA is completely trusted to validate the identity of an entity requesting a digital certificate. In some cases, this functionality may be delegated to other entities called "*trusted introducers*". The trusted introducers are subordinate CAs. In this manner, the validation process can be effected by each of the trusted introducers in an accelerated fashion. The trusted introducers cannot ordain other entities as trusted introducers. A similar process is used in PKI technology with X.509 certificates where the network of trusted CAs is established based upon the root CA's signature on subordinate certificates; any certificate signed by the CA is considered valid to others within the hierarchy. This process of checking back up through the system to see who signed whose certificate is called *tracing*

a *certification path or certification chain*. Now that a few new terms have been introduced, let's investigate the different types of trust models prevalent today.

a. Direct Trust

In this model, a user trusts that a key is valid because he knows where it came from. All cryptosystems use this form of trust in some way. For example, in web browsers, the root Certification Authority keys are directly trusted because they were shipped by the manufacturer. If there is any form of hierarchy, it extends from these directly trusted certificates. In PGP, a user who validates keys themselves and never sets another certificate to be a trusted introducer is using direct trust. [Ref. 13]

b. Hierarchical Trust

In a hierarchical system depicted in Figure 4.8, there are a number of "root" certificates from which trust extends. These certificates may be used to certify certificates themselves, or they may certify certificates that certify still other certificates down some chain. Consider it as a big trust "tree". The "leaf" certificate's validity is verified by tracing backward from its certifier, to other certifiers, until a directly trusted root certificate is found. [Ref. 13]

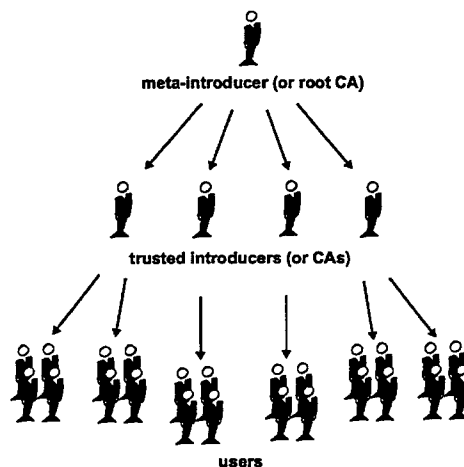


Figure 4.8 Hierarchical Trust. From Ref. [13].

c. Web of Trust

This model is the PGP view of trust. This model is a cumulative trust model where a certificate might be trusted directly, or trusted in some chain going back to a directly trusted root certificate or a trusted introducer. PGP uses digital signatures as its

form of introduction. When any user signs another's key, that user becomes an introducer of that key. As this process goes on, it establishes a web of trust.

In a PGP environment, any user can act as a CA. Any PGP user can validate another PGP user's public key certificate. However, such a certificate is only valid to another user if the relying party recognizes the validator as a trusted introducer. (That is, you trust my opinion that another person's key is valid only if you consider me to be a trusted introducer. Otherwise, my opinion is moot.) [Ref. 13]

Stored on each user's public keyring are indicators of:

- whether or not the user considers a particular key to be valid
- the level of trust the user places on the key that the key's owner can serve as certifier of other keys

You indicate, on your copy of my key, whether you think my judgment counts. It is really a reputation system: certain people are reputed to give good signatures, and people trust them to attest to other keys' validity. [Ref. 13]

5. Levels of Trust in PGP

The highest level of trust in a key, implicit trust, is trust in your own key pair. PGP assumes that if you own the private key, you must trust the actions of its related public key. Any keys signed by your implicitly trusted key are valid.

There are three levels of trust you can assign to someone else's public key:

- Complete trust
- Marginal trust
- No trust (Untrusted)

To make things confusing, there are also three levels of validity:

- Valid
- Marginally valid
- Invalid

To define another's key as a trusted introducer, you:

1. Start with a valid key, one that is either
 - signed by you or
 - signed by another trusted introducer

and then

2. Set the level of trust you feel the key's owner is entitled.

For example, suppose your keyring contains Alice's key. You have validated Alice's key and you indicate this by signing it. You know that Alice is a real stickler for validating keys. You therefore assign her key with *Complete* trust. This makes Alice a CA. If Alice signs someone else's key, it appears as *Valid* on your keyring. PGP requires one *Completely* trusted signature or two *Marginally* trusted signatures to establish a key as *Valid*. PGP's method of considering two *Marginals* equal to one *Complete* is similar to a merchant asking for two forms of ID. You might consider Sally fairly trustworthy and also consider Bob fairly trustworthy. Either one alone runs the risk of accidentally signing a counterfeit key, so you might not place complete trust in either one. However, the odds that both individuals signed the same phony key are probably small. [Ref. 13] For a more comprehensive look at setting a level of trust, see Ref. [17].

6. Certificate Revocation

The basic premise behind certificate revocation introduced in Chapter II is germane under PGP. Suppose both your secret key and pass phrase are compromised. It is now necessary to notify other users so that the compromised public key is no longer used for digital signature verification; the reason is because someone else may be able to impersonate you. A "key compromise" or "key revocation" certificate must be issued to revoke the compromised public key. Figure 4.9 illustrates the generation of a revocation certificate via the "pgp -kd" (permanently revoke your own key, issuing a key compromise certificate) command. A user attempting to sign a file with his revoked private key will be disallowed once the public key is revoked.


```
C:\pgp262>pgp -kd doe
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 2000/01/19 21:54 GMT

Key for user ID: John T. Doe <doe@cs.nps.navy.mil>
1024-bit key, Key ID 67468829, created 2000/01/19

Do you want to permanently revoke your public key
by issuing a secret key compromise certificate
for "John T. Doe <doe@cs.nps.navy.mil>" (y/N)? y

You need a pass phrase to unlock your RSA secret key.
Key for user ID "John T. Doe <doe@cs.nps.navy.mil>"

Enter pass phrase: Pass phrase is good. Just a moment....
Key compromise certificate created.
```

Figure 4.9 Creating a Revocation Certificate.

This revocation certificate bears your signature and is made with the same key you are revoking. By using the same key, it will replace the identical revoked key on the key server and on any user keyring to which it is added. You should widely disseminate this key revocation certificate as soon as possible preferably through the "bamboo-users" mailing list and by extracting the key and storing it on the key server. Others who receive it can add it to their public keyring, and their PGP software then automatically prevents them from accidentally using your old public key ever again. You can then generate a new secret/public key pair and publish the new public key. In Figure 4.10, the revoked key is extracted into an ASCII text file and printed to the screen.

```
C:\pgp262>pgp -kxa doe
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 2000/01/19 22:03 GMT
```

Extracting from key ring: 'pubring.pgp', userid "doe".

```
Key for user ID: John T. Doe <doe@cs.nps.navy.mil>
1024-bit key, Key ID 67468829, created 2000/01/19
Key has been revoked.
```

Extract the above key into which file? **revoke**

Transport armor file: revoke.asc

Key extracted to file 'revoke.asc'.

```
C:\pgp262>type revoke.asc
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: 2.6.2
```

```
mQCNAziF1rwAAAEANLxAXdRBtxe0fpRdLhxS0dfv81YWY7B1oMYQAwoIOc4SK0
TiLKpqrnotCywRGsPI5C49IKXvda+d6jXiALLdHjdCJ008hWh4t3HXMiBXe+Cel7
CXSTpoP3cM9egp06zxMCzZxzFKN7dY11DiFPkpluyy&dZDmhZxhL4BnRognAAUR
iQCVAwUgOIYypZxhL4BnRognAQEIfAP/YfqhlaWTI8KJhA9xXxBBE+/BymNz+kyd
laZpjBXkA9NHHHAFB1Cva9XbZPBmKJZnMDQdAm6oWi+UqVRCSiLa8EHrP+AQd8L
k2QSSXqSSA1nqgXRVyRBvLBkW9tk6g0AbxMoYdzvhLs8sqthXQfCO6KgU4aLe+CBK
IOIMv3C/ziG0IUpvaG4gVC4gRG9IIDxkh2VAY3MubnBzLm5hdkubWlsPg==
=sPH8
-----END PGP PUBLIC KEY BLOCK-----
```

Figure 4.10 Extracting and Printing the Revocation Certificate.

But what happens if the user can no longer access his private key or if the CA no longer backs the userID/public key combination on the PGP certificate? With PGP v2.6.2, the only recourse is to notify as many users as possible that the public key is no longer valid and should be disabled on their public keyrings (Figure 4.11). A disabled key may not be used to encrypt any messages, and may not be extracted from the keyring with the "pgp -kxa" command. However, it is important to point out that it can still be used to check signatures, therefore it is incumbent upon the user to be aware of the global status of all keys on his keyrings.

```
C:\pgp262>pgp -kd smith
Pretty Good Privacy(tm) 2.6.2 - Public-key encryption for the masses.
(c) 1990-1994 Philip Zimmermann, Phil's Pretty Good Software. 11 Oct 94
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 2000/01/19 22:14 GMT

Key for user ID: Robert E. Smith <smith@cs.nps.navy.mil>
1024-bit key, Key ID CA5AA3E5, created 2000/01/19

Disable this key (y/N)? y
```

Figure 4.11 Disabling a Public Key on your Keyring.

In the case of Bamboo, one potential mechanism to revoke a user's public key is to decertify the public key certificate stored on the key server and notify all users via the "bamboo-users" mailing list. An end-user downloading a signed archive module would need to ensure that a valid key is still on the key server; this will ensure the end-user is receiving a verifiable signed archive module.

7. Storing and Extracting Keys at the Bamboo PGP Public Key Server

The Bamboo PGP Public Key Server is discussed in detail below. Its primary purpose is to serve as a repository for the distribution and management of public keys. End-users intending to sign archive modules should store their public key on the key server. This enables other end-users downloading the signed archive modules to perform a digital signature verification.

a. Storing a Public Key on the Key Server

To store a public key on the key server, it must first be extracted from the user's public keyring according to procedures outlined in Figure 4.2. Copy the key information from the ASCII file as in Figure 4.3 and paste the material into the key server web page under the heading *Submit a Bamboo PGP public key to the server*. (Currently, only available in-house (<http://hendrix/~smithml/KeyServer.html>) but will eventually migrate to the Bamboo website at Ref. [19]). Depress the *Submit* button and you have added a public key to the server for others to access. See Figure 4.12.

Submit a Bamboo PGP public key to the server

- Here's how to add a Bamboo PGP public key to the server's database:
 1. At the command line, execute: `pgp -kxa userid filename` to extract a public key to a file in ascii format (see PGP user's guide for details).
 2. Using your favorite text-editor, open the file created in previous step and cut-and-paste the ASCII-armored version of your public Bamboo key into the text box.
 3. Press "Submit".
- The key server will process your request immediately. If you like, you can check that your Bamboo key exists using the extract procedure above.

Enter ASCII-armored Bamboo PGP key here:

-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: 2.6.2

mQCNAzIGQBKA A A EEA MzAUkZabAUWMEWqSBYm0G6
+E6DdXoW SxaqrBdadnCR63
DT0A6/zYuNeXNx CJty2NwJeDleqddV33V uUd1bBN3DTITL8PhQuh
ONOCYruGvk1
gMUIKGv8nxgx8Np7SSH1MQJL3Ok4eh6TOS9V10benoH3aENXKlrN7
KWqPIAAUR
tCdSb2JlenQgRS4gU21pdGggPHNtaXR0QQNzLm5w oy5uYXZ5Lm1p
D4=
=BQzh
-----END PGP PUBLIC KEY BLOCK-----

Reset Submit this key to the Bamboo key server!

Figure 4.12 Subset of Key Server Web Page for Storing Key.

b. Extracting a Public Key from the Key Server

In order for a user to extract a public key, first access the key server web page via the link above. Scroll down to the heading *Extract a Bamboo PGP public key from the server*. Either type the userID of the desired key, or if unsure of the userID, type a portion of the userID or name. Depress the *Do the search!* button and the key server will perform a pattern match. Once the key server lists all of the "hits", the user can click on any of the returned hits to view the public key information. Upon locating the desired key's information, it should be copied into a text editor and saved with a '.asc' extension. Follow the procedures above under paragraph *Storing a Public Key on a Keyring* to place the key onto the public keyring. Now the extracted key should be verified and is subsequently ready to use for digital signature verification. See Figures 4.13, 4.14.

Extract a Bamboo PGP public key from the server

- Here's how to extract a Bamboo PGP public key:
 1. Select either the "Index" or "Verbose Index" check box. The "Verbose" option also displays all signatures on displayed keys.
 2. Type ID you want to search for in the "Search String" box.
 3. Press the "Do the search!" key.
 4. The server will return a (verbose) list of keys on the server matching the given ID. (The ID can be any valid argument to a `pgp -kv(v)` command. If you want to look up a key by its hexadecimal KeyID, remember to prefix the ID with `0x`.)
 5. The returned index will have hypertext links for every KeyID, and every bracket-delimited identifier (i.e. `<smithml@cs.nps.navy.mil>`). Clicking on the hypertext link will display an ASCII-armored version of the listed public key.

Index: ☒ Verbose Index: ☐

Search String:

☒ Show PGP "fingerprints" for Bamboo keys

☐ Only return exact matches

Figure 4.13 Subset of Key Server Web Page to Extract Key.

Public Key Server -- Index "smith"

Type bits/keyID	Date	User ID
pub 512/0481F411	2000/01/26	*** KEY REVOKED *** Marlon L. Smith <smithml@cs.nps.navy.mil>
Key fingerprint = A1 63 8A 10 17 8A 82 1D CC 3C 6D FC 9F B3 3D 28		
pub 1024/D749EAD5	2000/01/27	Marlon Smith <smithml@cs.nps.navy.mil>
Key fingerprint = 89 AC FC 9A C4 B7 04 19 A8 44 59 44 91 C4 5B F7		

Figure 4.14 Result from Key Server Search.

B. USING PGP WITH BAMBOO

Upon completing the installation procedures for PGP v2.6.2, Bamboo modules may be digitally signed. Bamboo indirectly calls PGP using Netscape Portable Runtime (NSPR) [Ref. 19]. In addition to PGP, the code required to perform the module signing is located primarily in two files. The files, `bbModule.c++` and `archiveApp.c++`, are not included in this thesis but are available in the Bamboo software download. (See Ref. [19].)

The module signing application `archiveApp.c++` is a standalone utility application in the `Utils` directory under the `Bamboo` directory. It is executed via the `archive.bat` file in the `Bamboo` directory. It is menu-driven and offers the following menu options:

- Archive/sign a module or sign a previously archived module.
- Removing a signature from an archive module.

- Display signatures on a signed archive module.
- Exit.

Since module signing is an optional feature, a user may choose to archive a Bamboo module and not sign it. This is the only option if PGP is not installed.

The bbModule.c++ software reads archive modules of varying formats (listed below), provides digital signature verification if required, and expands the archive module into its original directory structure in the user's '.bamboo/cache' directory.

1. Bamboo File Formats

To provide the reader with insight as to the data requirements necessary for this application, the internal module file formats are illustrated. The following figures provide a snapshot of the internal format of different files processed by Bamboo. Specifically, Bamboo reads archive modules pre-dating module signing (old archive format with '.tar.gz' extension), unsigned archive modules, and signed archive modules consisting of one or more signatories. Reviewing the structure of the different file formats will assist in the discussion of the logic flow in a later section. Figure 4.15 illustrates the internal format found in '.tar.gz' files pre-dating this security implementation. Figures 4.16 and 4.17 illustrate the file formats created from object signing in Bamboo. The first consists of a signed module and the second is an unsigned module; both are stored as '.bar' files and the software determines which type of file it is.

<u>Field Identifier</u>	<u>Description</u>	<u>Byte Positions</u>
Data	Archive data.	Variable Length starting at 0.

Figure 4.15 File Format for Old Archive Module (non-PGP
and '.tar.gz' format).

<u>Field Identifier</u>	<u>Description</u>	<u>Byte Positions</u>
Header	"HDFMT V1.0.0n"	0 - 13
File Type	'S' = Source, or 'O' = Object, or 'D' = Data followed by 'n'.	14 - 15
No. of Signatures	String representing integer ending in '\0', i.e., "10".	Variable length starting at 16
*UserID #1	String ending in '/'; i.e., "John T. Doe <doe@cs.nps.navy.mil>/'".	
*Hash Algorithm	"1n"; 1 = MD5. For future use.	
*Hash #1 Length	String ending in 'n'.	
*Hash #1	Digital Signature followed by '/	
Data	Archive data.	

* Repeat for signed archive modules with more than one signature.

Figure 4.16 File Format for Signed Archive Module (PGP).

<u>Field Identifier</u>	<u>Description</u>	<u>Byte Positions</u>
Header	"HDFMT V1.0.0n"	0 - 13
File Type	'S' = Source 'O' = Object 'D' = Data	14
Zero Signatures	"n0n"	15 - 17
Data	Archive data.	

Figure 4.17 File Format for Unsigned Archive Module.

2. Source/Destination Directory of Archive Module

The primary repository for storage and access of previously archived modules is the '.bamboo/cache' directory (specified during Bamboo initialization). A referenced archive module is usually located in the '.bamboo/cache' directory for access. A user may specify just the module name, e.g. *myModule*, and the application will "name mangle" (attach extensions) based upon operating system and search for the appropriate module name. For instance, if using SGI Irix, then it would search for *myModule.sgi.bar*. The user may also specify file names via direct reference such as *c:/bamboo/cache/myModule*. Conversely, when specifying a module for initial archive,

the module should be under the Bamboo directory and is searched recursively inspecting subdirectories for the module name.

3. Potential Problems

- In this security implementation, the software checks for content integrity of a signed archive module prior to decompression and unarchiving. If the user does not possess the public key for signature verification or if content integrity problems are encountered, then the user will be notified. The user is still afforded the opportunity to decompress and unarchive the module, yet is cautioned against such an action.
- The user must ensure the existence of the '.bamboo/cache' directory prior to signing or archiving a module. This should not pose a problem since it is dynamically created via Bamboo if not already present.
- During archive and signing operations, the source directory or file will be preserved. However, there are various file management operations issued during these processes. If the software encounters an error, an appropriate error message will be displayed with the necessary means to correct the situation.

4. Archiving a Module

a. Setting Permissions on a Module Directory Structure

In order to properly archive a module (directory structure), the user must ensure proper permissions are enabled on the files and the parent directory to archive. That is, under Unix, if permissions are not set to *Read* then the *archive* command will fail and the user notified.

b. Process of Archiving and Signing a Module

The *archive* command accepts a module (specified directory) and archives all files under the directory (including sub-directories) using the familiar Unix "tar" operation. The intermediate archive module will have the name: "file_name.platform_type.tar". The next operation executed is the familiar "zip" which compresses the module to a suitable size for transport. The resulting intermediate archived and zipped module is then of the name format "file_name.platform_type.tar.gz".

This file is equivalent to the archive module format under the previous Bamboo implementation.

The user is also prompted to identify the module type. The code inspects the file content of the '.tar.gz' file to determine if its contents appear to match the selected type based upon specific search criteria. If the software determines a possible incorrect type specification, it will prompt the user to confirm the type specification.

The next step is to optionally sign the module. If the user decides not to sign the module, it is saved as an unsigned archive module in the '.bamboo/cache' directory with the name format "file_name.platform_type.bar" (see Figure 4.18). If the user chooses to sign the module, the user's private keyring is accessed and a list of private key userIDs is displayed. Upon selection of a private key, the user enters his password to digitally sign the module. By signing a module, the software hashes the current '.tar.gz' module and encrypts the hash using the RSA encryption algorithm. (PGP v2.6.3i uses a compatible encryption algorithm due to an RSA patent.) The encrypted hash, userID, and other pertinent data are inserted into the intermediate '.tar.gz' (see Figure 4.16). Once module signing is complete, the file is again saved in the '.bamboo/cache' directory with the name: "file_name.platform_type.bar". Figure 4.18 is a sample script of a user signing a module.

hendrix 41% archive

**** Bamboo Archive Module Menu: (select 1-4) ****

(Note: previously archived modules must be in .bamboo/cache dir)

- 1.) Display list of signatures attached to archived module.
- 2.) Remove a signature from an archived module.
- 3.) Archive/sign a module or sign a previously archived module.
- 4.) Exit.

: 3

Enter the [path/]name of the module : **myModule**

Enter type of file to archive (s=source, o=object, d=data) : **o**

Module tarred and zipped as:

/worka/smithml/.bamboo/cache/myModule.sgi.bar

Do you desire to sign module with your PGP private key? (y/n) : **y**

List of user id's:

1. Marlon L. Smith <smithml@cs.nps.navy.mil>

Name (userid) to attach to this archive module:

"Marlon L. Smith <smithml@cs.nps.navy.mil>"

You need a pass phrase to unlock your RSA secret key.

Key for user ID "Marlon L. Smith <smithml@cs.nps.navy.mil>"

Enter pass phrase:

Key for user ID: Marlon L. Smith <smithml@cs.nps.navy.mil>

512-bit key, Key ID B06B9FF5, created 2000/01/25

Signed archive module is:

/worka/smithml/.bamboo/cache/myModule.sgi.bar

Figure 4.18 Sample Script of a User Signing a Module.

c. Logic Diagram for Archive and Signing Operation

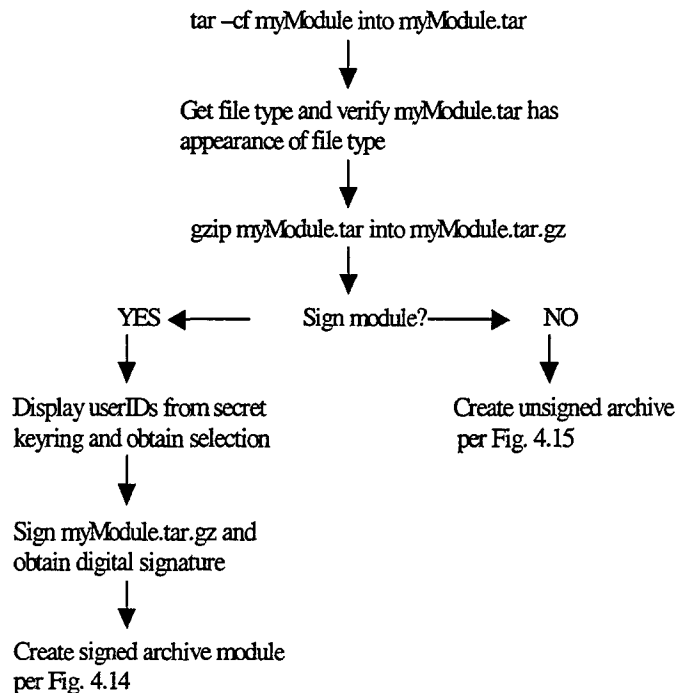


Figure 4.19 Logic for Archive and Signing Operation.

5. Signing a Previously Signed Archive Module

Previously, the *Web of Trust* was defined as a chain of PGP users trusting a public key as long as somewhere in the web, there exists a single trusted key. The same cumulative trust concept applies to signed archive modules. For instance, let's say User B obtains a module originally signed by User A, verifies the digital signature, and performs testing with the module. After extensive testing, User B decides that he "trusts" this module and wishes to sign it also (refer to Fig 3.3). Furthermore, say User C downloads the module to perform testing; yet, User C does not know the identity of User A but trusts User B. The motivation for allowing multiple signatories is easily perceived from this example. By allowing multiple signatures on an archive module, a *Web of Trust* is formed around the signatures attached to a module.

a. Process of Signing a Previously Archived Module

If a `.bar` extension is specified on the module name, then the software will check the `.bamboo/cache` directory to locate the file. The header will be inspected for compliance with the new PGP format. If the module is of the old format (`.tar.gz` extension), then the user must unarchive the module first via the *bamboo* command.

Once unarchived, the module can be archived and signed directly into the new format as specified in Figure 4.16. If the module is in the latest PGP format, then all userIDs attached to the module will be displayed. The user will then be prompted to select his private key for digital signing and a new signed archive module will be saved in the '.bamboo/cache' directory with his signature attached.

Figure 4.20 is a sample script of a user signing a module.

```
hendrix 43% archive

** Bamboo Archive Module Menu: (select 1-4) **
(Note: previously archived modules must be in .bamboo/cache dir)

1.) Display list of signatures attached to archived module.
2.) Remove a signature from an archived module.
3.) Archive/sign a module or sign a previously archived module.
4.) Exit.
: 3

Enter the [path/]name of the module : myModule.sgi.bar

Module currently signed by:
1. "Marlon L. Smith <smithml@cs.nps.navy.mil>"

Continue? (y/n) : y

List of user id's:
1. John Q. Public <public@cs.nps.navy.mil>

Name (userid) to attach to this archive module:
"John Q. Public <public@cs.nps.navy.mil>"

You need a pass phrase to unlock your RSA secret key.
Key for user ID "John Q. Public <public@cs.nps.navy.mil>"

Enter pass phrase:

Key for user ID: John Q. Public <public@cs.nps.navy.mil>
1024-bit key, Key ID 74440BED, created 2000/01/25

myModule.sgi.bar in /worka/public/.bamboo/cache/ is signed!
```

Figure 4.20 Sample Script of a User Signing a
Previously Signed Module.

b. Logic Diagram to Sign a Previously Archived Module

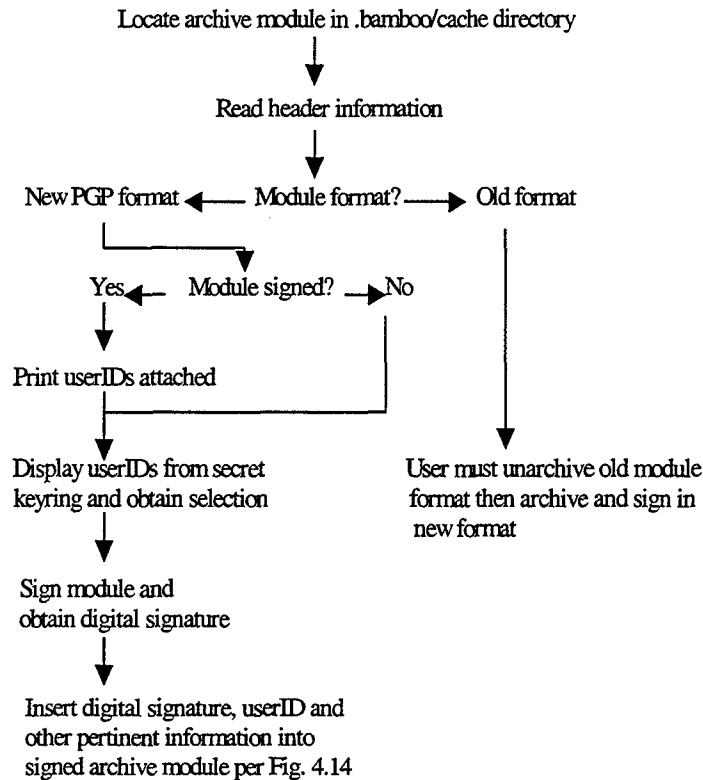


Figure 4.21 Logic to Sign a Previously Archived Module.

6. Display Signatures (userIDs) on a Signed Archive Module

Modules downloaded from a server (or received via other means) and placed in the '.bamboo/cache' directory should be viewed to identify the list of userIDs attached. By previewing the names, the user can determine whether he possesses required public keys on his keyring for signature verification. If necessary keys are not present, then extracting the requisite keys from the key server is the next step. The software is capable of unarchiving modules without signature verification but this practice is not recommended. Figure 4.22 is an example of a user displaying names attached to a signed archive module.

```

hendrix 43% archive

** Bamboo Archive Module Menu: (select 1-4) **
(Note: previously archived modules must be in .bamboo/cache dir)

1.) Display list of signatures attached to archived module.
2.) Remove a signature from an archived module.
3.) Archive/sign a module or sign a previously archived module.
4.) Exit.
: 1

Enter the [path/]name of the module : myModule.sgi.bar

Module currently signed by:
1. "John Q. Public <public@cs.nps.navy.mil>"
2. "Marlon L. Smith <smithml@cs.nps.navy.mil>"

```

Figure 4.22 Sample Script of Viewing Signatures Attached to a Module.

7. Removing a Signature (userID) on a Signed Archive Module

It may be desirable to remove a signature from a module. If a public key is revoked or disabled, then prudence would dictate removal so that there is no inherent trust associated with these signatures. Figure 4.23 illustrates the procedure.

```

hendrix 3% archive

** Bamboo Archive Module Menu: (select 1-4) **
(Note: previously archived modules must be in .bamboo/cache dir)

1.) Display list of signatures attached to archived module.
2.) Remove a signature from an archived module.
3.) Archive/sign a module or sign a previously archived module.
4.) Exit.
: 2

Enter the [path/]name of the module : myModule.sgi.bar

Module currently signed by:
1. "John Q. Public <public@cs.nps.navy.mil>"
2. "Marlon L. Smith <smithml@cs.nps.navy.mil>"

Select signature to remove from archive module (1 - 2, x to exit): 2

Confirm user ID signature to delete: "Marlon L. Smith <smithml@cs.nps.navy.mil>"
(y/n) : y

Signature removed from: myModule.sgi.bar

```

Figure 4.23 Sample Script of Signature Removal from a Module.

8. Storing a Signed Archive Module on a Server

a. *Setting Permissions on a Signed Archive Module*

Prior to storing a signed archive module on a server for mutual access by other users of a network, ensure the appropriate permissions are set for *Read* access.

b. *File Transfer Protocol (FTP) for Upload*

FTP is a resourceful utility to store a module from the '.bamboo/cache' directory to a server for distribution to other users. Modules shall be stored on the server with name mangling included.

9. Retrieving a Signed Archive Module via FTP

FTP is convenient for downloading modules from a server and storing locally in the '.bamboo/cache' directory. From here, it is easily processed by Bamboo.

10. Loading Archived Modules via Bamboo

a. *Dynamic Download of Archive Modules*

If running Bamboo and a module is not available locally, it is feasible to download a module dynamically followed by signature verification. This principle is similar to current browser technology for the download and installation of plugins.

Additionally, a user may download an archive module via the *bamboo* command followed by the uniform resource locator (url), i.e., *bamboo http://hendrix/~smithml/myModule*. Again, the software will perform required name mangling prior to sending the request to the server. Figure 4.24 illustrates.

```
hendrix 43% bamboo http://hendrix/~smithml/myModule
# Bamboo(c) v0.1215 (the "y2k" v1.0 alpha3) released on January 01, 2000

Names attached to module:
1. "John Q. Public <public@cs.nps.navy.mil>"

Good signature from user "John Q. Public <public@cs.nps.navy.mil>".
Signature made 2000/01/25 01:38 GMT

kernel Notice: A file has been downloaded and placed inside your Bamboo
kernel Notice: cache directory, ...
```

Figure 4.24 Sample Script using Bamboo to

Download a Module from a Server.

b. Logic Diagram for Archive and Signing Operation

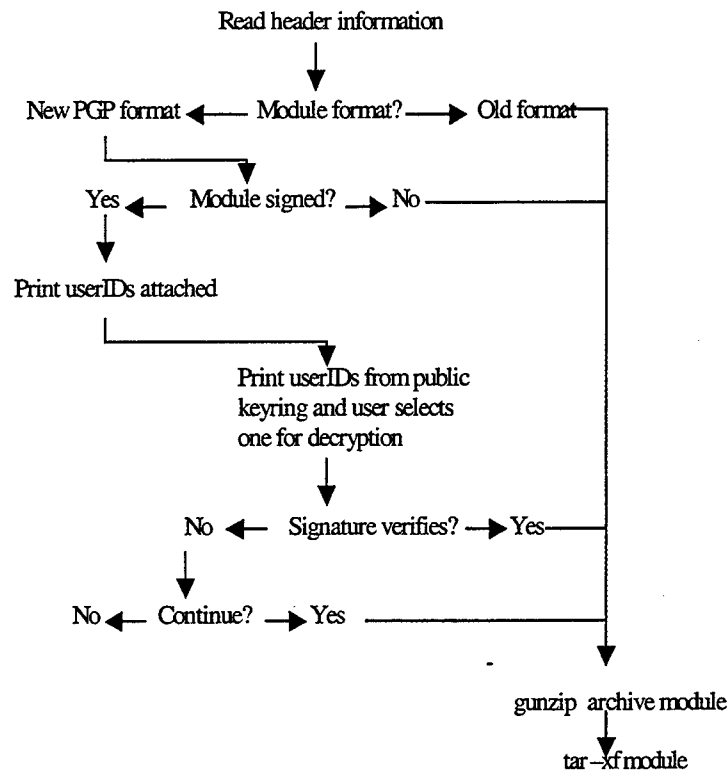


Figure 4.25 Logic for Signature Verification followed by Module Decompression and Unarchiving.

C. BAMBOO PGP PUBLIC KEY SERVER

The public key server system is a set of programs that manages and provides general access to a database of PGP public keys. The database itself is not a standard PGP keyring. Instead, the keys managed by the key server are stored in a set of Berkeley Database (DB) 2.x format database files. This key server is the identical system used in other PGP Public Key Servers worldwide and is available for download via Ref. [20]. These servers (called peer key servers) have been instituted to create a network of PGP key databases mirroring one another in content. Each of the servers communicate via e-mail to send the latest key updates to ensure data consistency.

In Bamboo, the key server has been incorporated as a single site database repository for the storage and retrieval of public keys used in object signing. If the object

signing feature is adopted by a large number of Bamboo users, investigation into establishing a mirror database site should be initiated.

Figure 4.26 displays the physical architecture of the key server. In the figure, there are numerous users accessing the key server to store their own public key on the server and to extract other users' public keys for insertion onto their personal public keyrings.

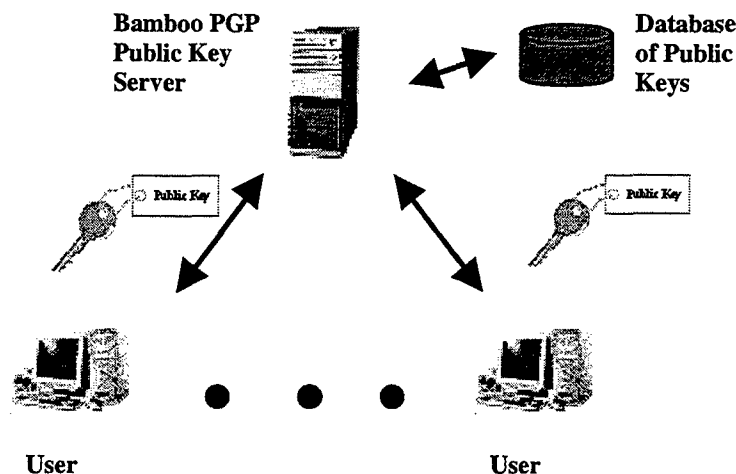


Figure 4.26 Physical Architecture of Bamboo PGP Public Key Server.

1. Bamboo PGP Public Key Server Architecture

The physical architecture above hides the actual implementation or logical architecture. The logical architecture consists of a group of key server programs including the server daemon, the server control program, and the server client program. There is a configuration file providing parameters for both the server daemon and the control program.

a. Configuration File (*pkd.conf*)

The *pkd.conf* configuration file contains all the information for the programs comprising the public key server system. The e-mail functionality is not incorporated at this time, therefore many of the configuration variables are not used. The configuration file variables are:

- *pks_bin_dir* – Directory location of executables which form the basis of the key server system.

- *db_dir* – Directory containing database files.
- *www_port* – Port number for accepting HTTP connections.
- *socket_name* – Unix domain socket in which the pksd program will listen for control messages from the pksdctl program to include notification of new mail messages.
- *mail_delivery_client* – Not used.
- *maintainer_email* – Not used.
- *mail_intro_file* – Not used.
- *help_dir* – Directory containing key server help files. The files in this directory should be named pks_help.LANG, where LANG is the language of the help file, in lower-case. These files are also used as MIME parts, so they are subject to the same formatting requirements as the mail_intro_file.
- *default_language* – Default language; uses English.
- *this_site* – Not used.

b. Server Daemon (pksd)

The pksd program is a daemon implementing the Bamboo PGP Public Key Server. It supports searches, key requests, additions, and modifications via the WWW and email interfaces (e-mail not instantiated at this time). The server reads the pksd.conf file for initial parameters.

The server may support queries via the web and an email interface. To control the pksd server daemon during execution, the pksdctl program is used to send messages via a Unix domain socket (see pksdctl below).

In addition to adding public keys and revocation certificates to the server database, the server must handle disabled keys. As previously stated, disabling a key is necessary when a userID requires unbinding from a key. This is a function that the server administrator or an individual user can perform. It is not useful to remove a disabled key from the key server database, since it will probably just disappear. Disabled keys cannot be retrieved from the database to users but are returned by searches.

c. Server Control Program (pkscctl)

The pkscctl program sends strings to an executing pkscd via a Unix domain socket. Each string is interpreted by the pkscd program as a command. Three commands are supported, namely: *mail*, *disable*, and *shutdown*. A description of each follows:

- *mail msg* - The file “msg” will be parsed as a mail server request. If the file is a valid request, the request succeeds, and the response can be enqueued, then the file is removed.
- *disable userID* - All keys matching “userID” are disabled.
- *shutdown* - The key server daemon is shut down.

d. Server Client Program (pkscclient)

The pkscclient program is a command-line interface for the administrator to perform key server operations directly instead of through the daemon. The software will use locking and transaction semantics unless specified otherwise. Bypassing the locking mechanism should be avoided since another process may be accessing the same database record. Once a command is completed, pkscclient will attempt to checkpoint the database and remove any excess log files (see Database Administration below).

Each command accepts the directory path of the database files, a command name, and optionally a list of arguments, i.e., *pkscclient /db/path create [num_files]*. Some commands take an optional flag argument. Flag arguments are discussed in Appendix B. The following comprise the commands for the pkscclient program:

- *create [num_files]* - Create an empty database overwriting an existing database if present.
- *recover* - Recover an inconsistent database.
- *add filename [flags]* - Add a keyring to the database.
- *get userID [flags]* - An ASCII-armored keyring containing all the keys matching the userID is printed to standard output (stdout).
- *index userID [flags]* - An index listing for all the keys matching the userID is printed to stdout.

- *since time [flags]* - An ASCII-armored keyring containing all the keys added to the database or changed since the Unix timestamp is printed to stdout. The timestamp that the database was last modified is printed to standard error (stderr).
- *delete userID [flags]* - All keys matching the userID are deleted from the database.
- *disable userID [flags]* - All keys matching the userID will have the disabled flag set.

2. Database Administration

To properly manage the public key database, an administrator needs to be identified and trained in the specifics of the database. The key server is based on the open-source Berkeley Database (DB); the DB includes full transactional support and database recovery, online backups, and separate access to locking, logging, and shared memory caching subsystems. Comprehensive documentation on the DB is available at Ref. [21].

Fortunately, the Berkeley DB has interfaced with the PGP Key Server software with exceptional results since 1996 and is employed worldwide at numerous PGP sites. However, there may be the occasion when a database crashes and procedures for archive (backup) and recovery require execution.

One key component for database restoration in the Berkeley DB infrastructure is performing checkpoints of the log files. As transactions commit to the database, records are written into the log files, but the actual changes to the database may not be written to disk. When a checkpoint is performed, the changes to the database that are part of committed transactions are written into the database.

Performing checkpoints is necessary for two reasons. First, you can only remove the log files from your system after a checkpoint. Second, the frequency of your checkpoints is inversely proportional to the amount of time it takes to run database recovery after a system or application failure.

Once the database pages are written, log files can be archived and removed from the system because they will never be needed for anything other than a catastrophic

failure. In addition, recovery after system or application failure only has to redo or undo changes since the last checkpoint, since changes before the checkpoint have all been flushed to the file system.

Archiving, recovery, and checkpointing are discussed further in Appendix B and Refs. [20, 21].

D. ESTABLISHING A SECURITY POLICY FOR PUBLIC KEYS

The Web of Trust is only as effective as all of the participants exercising the security application. It is imperative that vigilance is maintained throughout and each user routinely compares the latest key status on their public keyring against the corresponding keys at the key server. Additionally, if any key is suspect, the key server administrator needs notification to take appropriate action such as disabling a key.

An initial security policy for this implementation may appear as follows:

- The server administrator performs CA functions.
- Require all users storing keys on the key server to join the “bamboo-users” mailing list.
- All users storing a public key on the key server should also e-mail their key information, address, and telephone number to the server administrator and only the key information to the “bamboo-users” mailing list. In this manner, the server administrator can compare the e-mail address of the sender with that stored on the key server and ensure compliance with userID naming conventions. If non-compliant, disable the key on the server and notify the user to send a compliant key. Additionally, the server administrator can ensure the individual is a member of the “bamboo-users” mailing list.
- Based upon the level of activity of keys stored on the key server, determine an ideal time period to perform checkpointing to clean up the log files. Refer to Appendix B.
- Whenever a key is disabled either by a user or the server administrator, notify all personnel on the “bamboo-users” mailing list.
- Whenever a revocation certificate is issued on the key server, notify all users via the “bamboo-users” mailing list. This ensures maximum dissemination.

The security policy illustrated is rudimentary in nature and will need refinement as time elapses and weaknesses are identified. No matter how stringent the refined policy, its evolution must be weighed against the usability by end-users and benefits that it provides. Its overall effectiveness must be consistently evaluated from an objective perspective realizing that no security system can be made foolproof. The next chapter will cover risks associated with PKI technology by one of the renowned experts in Computer Security, Bruce Schneier.

THIS PAGE INTENTIONALLY LEFT BLANK

V. RESULTS

The goal of this thesis was stated in Chapter III in the Software Requirements Specification: "...receive a degree of content integrity and user authentication...". The term "degree" is subjective and may vary from reader to reader. In order to apply a standard of measurement to the term, the reader must understand the risks associated with the implemented security application. Once introduced, the reader can then make an assessment as to the degree of content integrity and authentication provided.

Before discussing the results of the PGP v.2.6.2 system, this chapter reviews vulnerabilities prevalent in a typical PKI system. One should not be misled into thinking the most expensive security system is foolproof and provides insurmountable safeguards. Such a system does not exist, as the following discussion will demonstrate. Following the discussion, risks associated with PGP v.2.6.2 are reviewed. Only by examining one's own security system and identifying the weakest areas will an administrator be able to improve the organization's intended assurance level.

A. RISKS IN A TYPICAL PKI SYSTEM

Trust is defined as: "To rely on or depend on confidently; to believe in another". [Ref. 22] When dealing with certificates, security is a chain of events. The security of any CA-based system is based on many links, not all cryptographic. People and computer systems are involved. Since people are not secure and work with computer systems, the computer systems are not secure.

Attacking a typical PKI system requires an individual adept at cryptanalysis or a technically savvy perpetrator who understands the underlying infrastructure of PKI technology. Since public keys are available to anyone, cryptanalysis can be performed remotely (beyond the firewall of the organization holding the private key) and is independent of the PKI system. On the other hand, to attack the PKI infrastructure, the perpetrator must be inside the organization's firewall ("inside job") or be able to circumvent some process between the CA and the organization.

1. Cryptanalysis Attack

A cryptanalysis attack is a computationally expensive process to render a private key useless. It is an attempt to determine the composition of a private key using the public key as data input. This method requires extraordinary computer resources and an indeterminable amount of time.

In 1999, an effort was successful in factoring a 512-bit number using approximately 300 fast SGI workstations and Intel Pentium PCs, mostly on nights and weekends, over the course of seven months. It is certainly reasonable to expect 768-bit numbers to be factored within a few years. The 512-bit factoring event is significant since most of the Internet security protocols use 512-bit RSA. Anyone implementing RSA should have moved to 1024-bit keys and should be thinking about 2048-bit keys today. [Ref. 14]

2. Attacking the PKI Infrastructure

Ellison and Schneier detail 10 risks associated with PKI technology in Ref. [23]; a select few are paraphrased below:

a. Trusted Certificate Authority

A CA is not granted a level of trust from any national government authority; it has merely gained a *perceived trust* by virtue of reputation, having a large user-base, advertising, or from general acceptance of its formalized Certificate Practice Statement (CPS). Yet, all of the CPSs in existence disclaim liability, effectively detaching any meaning from the certificate; nor does the issued certificate binding an identity to a key expressly grant permission to perform any kind of transaction, thus leaving all risks to the parties utilizing the certificate during a transaction.

b. Vulnerability of your Private Key

Usually private keys are stored on a computer, the same computer subject to viruses or other malicious programs. If the key is safe on the computer, it also needs to be accessible by only the user. If it is compromised and someone signs using the private key, then the owner may be subject to risks associated with non-repudiation. Under some digital signature laws, if your signing key has been certified by an approved CA, then the user is responsible for whatever is initiated via that private key. This is in direct contrast

to mail order/telephone-order rules affiliated with credit cards where the owner can typically receive a refund for any illegitimate purchases made by an offender.

c. Security of Verifying Computer

The verifying computer receives a certificate during a transaction, uses the public key of the CA (or one or more root keys in the list) to validate the integrity of the certificate, then processes the sender's request via the public key embedded in the certificate. If an attacker can add his own public key to that list, then he can issue his own certificates, which are treated exactly like legitimate certificates. Since the X.509 v3 format is standardized, the illegitimate certificate mimics a legitimate one except it contains a public key of the attacker. The only solution is to have a verifying computer invulnerable to tampering.

d. Binding Process

The CA needs to identify the applicant before binding the applicant's name to his public key. Often, credit bureaus are used as the source of information to validate the identity of the applicant. However, a credit bureau provides no proprietary information that the applicant could not provide himself, and on occasion provides misinformation. Meanwhile, having identified the applicant somehow, how does the CA verify that the applicant really controls the private key corresponding to the public key being certified? Some CAs do not even consider this to be part of the application process. Could a perpetrator possessing my personal information request a certificate under my name?

e. Certificate Practice Statement

Certificates are an instrument used to assist in security implementation and are governed by the X.509 v3 standard. However, CAs establish their own CPSs and they vary widely. By accepting a specific CA's certificate, you are consenting to their CPS which may exclude the use of Certificate Revocation Lists or other commonly implemented methods for determining revoked certificates.

One of the greatest risks resulting from PKI solutions is the false sense of security perceived by organizations. Systems are typically installed to gain the confidence of customers conducting business with the organization by projecting a security conscious

theme. These systems are often labeled as minimal-impact “turnkey” security solutions and the requisite personnel are not always allocated for proper management. Despite their wide deployment, system administrators typically do not understand the underlying system, and PKI vendors are not in the business of advertising weaknesses in their respective technology. These PKI systems will continue to be susceptible to attacks unless the technology is fully understood and appropriate countermeasures implemented.

B. RISKS ASSOCIATED WITH PGP v2.6.2

PGP v2.6.2 can be circumvented in a variety of ways. Potential vulnerabilities include: compromising your pass phrase or secret key; public key tampering; deleted files that still physically reside on the disk; viruses; breaches in physical security; electromagnetic emissions; exposure on multi-user systems; traffic analysis; and even direct cryptanalysis. [Ref. 16]

1. PGP Vulnerabilities

A few of the vulnerabilities inherent with PGP are presented to introduce some limitations of the implemented system and to encourage proactive behavior in countering these potential threats. Those of larger scope such as physical security breaches, tempest attacks, and traffic analysis are left for the reader to investigate. The following information is courtesy of Zimmerman [Ref. 17]:

a. Compromised Pass Phrase and Secret Key

A compromise enables an individual to sign objects with your private key resulting in identical risks discussed in “Vulnerability of your Private Key” above. Bottom line: “Memorize your pass phrase and ensure your private key is not accessible by anyone”.

b. Public Key Tampering

Public key tampering is probably the greatest vulnerability and the most difficult to recognize. When you use someone's public key, make certain it has not been tampered with. A new public key should only be trusted if received directly from its owner or if signed by someone you trust. Additionally, this requires that your public keyring is not available to others and under your physical control. It is also suggested to maintain a backup copy of both keyrings. An example from Ref. [17] illustrates the

pitfalls of public key tampering with respect to PGP applications pertaining to encryption of e-mail; the following is tailored to object signing. Following the example, measures to prevent such a catastrophe are presented:

“Suppose you download a signed object code file from a server with Alice’s userID attached. You trust Alice as a person but unfortunately she is on a business trip. You download her public key from the key server so that you can decrypt the signature attached to the downloaded file for an authenticity and integrity validation check. You notice that the extracted public key is not signed by anyone certifying it as a valid public key. However, you decide to attach it to your public keyring anyway to perform the digital signature verification. Alice’s signature verifies on the downloaded file so you start executing the object code and subsequently all your files are erased from the hard drive. What went wrong?”

First, the public key extracted from the key server was not certified by the key server administrator or some other trusted introducer. Second, a perpetrator (most likely inside the organization with access to the key server and the server holding the signed files) stored a public key on the key server impersonating Alice. Additionally, he generated a phony private key with Alice’s name and used it to sign a virus file, then stored it on the server for others to download.

There are a couple of ways to prevent such a disaster. The first technique is to receive the public key directly from Alice and verify the key. The second technique is to obtain Alice’s key from a mutually trusted introducer who knows they have a valid copy of Alice’s key. Of course, you would need the public key of the introducer to validate his signature on Alice’s key. Utilizing the second technique is called “verification of a signed public key certificate”. This example illustrates the importance of having a trusted introducer such as the key server administrator to perform the public key verification followed by certificate signing as introduced in Chapter IV.

PGP keeps track of which keys on your public keyring are properly certified with signatures from introducers that you trust. All you have to do is tell PGP which people you trust as introducers, and certify their keys yourself. PGP can take it from there, automatically validating any other keys that have been signed by your

designated introducers. As introduced previously, you may directly sign more keys yourself. [Ref. 17]

Since your own trusted public key is used as a final authority to directly or indirectly certify all the other keys on your keyring, it is the most important key to protect from tampering. To detect any tampering of your own ultimately-trusted public key, PGP can be set up to automatically compare your public key against a backup copy on write-protected media.

PGP generally assumes you will maintain physical security over your system and your keyrings, as well as your copy of PGP itself. If an intruder can tamper with your disk, then in theory he can tamper with PGP itself, rendering moot the safeguards PGP may have to detect tampering with keys. [Ref. 17]

Bottom line: "No matter how tempted you are, never give in to expediency and trust a public key unless it is signed by someone you trust or has been personally verified".

c. Viruses and Trojan Horses

Another insidious attack involves a specially-tailored hostile computer virus or worm that may infect PGP or your operating system. This hypothetical virus could be designed to capture your pass phrase or secret key then covertly write the captured information to a file or send it through a network to the virus's owner. It may also alter PGP's behavior so that signatures are not properly checked. This attack is cheaper than cryptanalysis attacks.

There are some moderately capable anti-viral products commercially available, and following proper hygienic procedures can greatly reduce the chances of viral infection. PGP has no defenses against a virus, and assumes your own personal computer is a trustworthy execution environment. If such a virus or worm actually appeared, the whole PGP environment would have to be reinitialized by issuing revocation certificates on current keys, generating new key pairs, and signing all of the files again. [Ref. 17]

Another similar attack involves someone creating a clever imitation of PGP that behaves like PGP in most respects, but does not work the way it is supposed to.

For example, it might be deliberately crippled to improperly check signatures or allow bogus key certificates to be accepted. This "Trojan horse" version of PGP is not hard for an attacker to create since PGP source code is widely available. To prevent such an attack, every new version of PGP comes with one or more digital signatures in the distribution package, signed by the originator of that release package. Check the signatures on the version that you get and obtain copies from a reliable source such as MIT. [Ref. 17] Bottom line: "Ensure you have a valid copy of PGP and maintain a hygienic computing environment using the latest anti-viral software".

d. Exposure on Multi-user Systems

Local area networks are typically designed to allow users to view remotely located files such as your public and private keyrings. Additionally, there is a greater risk of involuntarily disclosing your pass phrase through special software to covertly monitor keystrokes. Recognize these risks on multi-user systems adjusting your expectations and behavior accordingly. Perhaps PGP should only be run on an isolated computer and signed files transferred to and from the multi-user system via magnetic media. Bottom line: "Be aware of the risks associated with multi-user systems and weigh the possible consequences of your actions".

In summary, understanding the intricacies of PGP is the first line of defense against malicious intent. Having previewed some of the vulnerabilities, most can be avoided by not succumbing to shortcuts. By following reasonable precautions, an attacker will have to expend far more effort and expense to violate your privacy. If you protect yourself against the simplest attacks, and you feel confident that your privacy is not going to be violated by a determined and highly resourceful attacker, then you are probably reasonably safe.

C. RESULTS OF PGP v2.6.2 IMPLEMENTATION IN BAMBOO

Safeguards to counter threats listed above can be defined via the security policy of the CA or by the organization using the certificates. The primary threats pertaining to the CA include the threat of public key tampering and procedures for binding of public keys--each are related. Other threats can be avoided at the user-level by exercising appropriate precautions that should be defined by the organization's formalized security policy.

The example under “*Public Key Tampering*” illustrates measures to prevent such a security violation. These measures are also outlined in Chapter IV under “ESTABLISHING A SECURITY POLICY FOR PUBLIC KEYS” where the key server administrator is the sole trusted authority on binding of public keys in the database through certification via his private key. By establishing appropriate guidelines on key storage, validation, and extraction at a key server, users can extract keys with the confidence that the key is valid and will not inflict pernicious acts against the user.

By understanding the weaknesses of PGP v2.6.2 and exercising proper precautions, end-users can obtain content integrity and user authentication subject to the key certification procedures of the key server administrator. Additionally, this functionality is at no monetary cost to the end-user and with minimal training. The extensive documentation on PGP permits the user to rapidly employ digital signing of files with instantaneous generation of key pairs.

In addition to the freeware aspect of PGP v2.6.2, is the freeware download of the key server software (available at Ref. [20]). At first glance, the reader may not glean the importance of the key server software since the Bamboo PGP Public Key Server is available for general use. However, if an organization desires to implement the security package within the confines of their organization (behind their firewall), they can institute their very own key server. This is advantageous for an organization desiring to establish their own CA and security policy for key binding and validation.

Along with the key server software, there is an active key server technical users’ group mailing list that provides prompt responses to any questions pertaining to the key server. This is a worldwide user group and consists of members utilizing the key server software implemented at various peer sites. Simply access the website at Ref. [20] and select the link “pgp-keyserver-folk@flame.org” to subscribe.

Overall, the security application meets the initial requirements set forth in the Software Requirements Specification. With further refinement of the security policy and feedback from the open-source community, the overall utility of the security application can be enhanced.

The concluding chapter discusses recent changes in export laws and possible ramifications of these changes. Chapter VI also provides areas of further research and the potential impact on current users if the PGP v2.6.2 system was replaced with another vendor's product.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

This thesis implements a simple architecture providing content integrity and user authentication to Bamboo users. Since the code is predominantly segregated from the core of Bamboo, it can be easily extracted from the Bamboo download and adapted to other applications. It utilizes the most widely used type of public key certificates, namely PGP. It is easy to implement and does not pose an encumbrance upon those not employing the security application.

B. SIGNIFICANCE

This thesis is resourceful in educating the reader in current PKI technology. It dispels the claim that PKI solutions are a security panacea by demonstrating vulnerabilities inherent in such systems. By presenting these vulnerabilities, it not only informs but also arms the user with the necessary tools to counter malicious intent by a perpetrator.

Many novice users have fueled the exponential growth in the Internet. In a typical file download, these users rely strictly on anti-viral software as the primary means to ensure a download is safe. However, this does not provide the assurance that the downloaded content has not changed or that the download site is whom they claim to be. In the commercial world, business-to-business (B2B) is the current buzzword for many of the e-commerce applications between organizations. Some B2B applications require the delivery of large files. Current bandwidth limitations and the expense of sophisticated high-bandwidth technology preclude many small to medium size businesses from reaping the full benefits of B2B e-commerce. Many still rely on ground or air transport for delivery of such files. With the increasing deployment of Digital Subscriber Line (DSL) and cable, bandwidth for large file delivery across the Internet will become commonplace. All organizations, large and small, will have the opportunity to participate in such transactions at a reasonable cost. Furthermore, with the continued emergence and acceptance of object signing technology, digital signing may become the standard for file transfers across the Internet.

C. FUTURE WORK

Rapid changes in technology and the recent revision of export laws play an integral part in the area of future work of object signing technology. This section reviews the recent export law changes, discusses licensing, and relates these issues to PGP. The final portion investigates potential products that may be used to replace the current security implementation to enhance the overall functionality, and discusses the impact on current users.

1. Export Law Changes

If Bamboo was strictly for users within the U.S. then the change in cryptography export laws would not be a major factor, however, such is not the case. During the period of this research, the Cryptsoft website (<http://www2.psy.uq.edu.au/~ftp/Crypto>) provided a freeware version of SSLeay but recommended consultation with a lawyer due to the export laws that were in force. With the relaxed restrictions, many systems mired in the legal web may be explored as potential candidates for a future upgrade.

A significant milestone in cryptography is the recent passage of revised export control laws. Effective January 12, 2000, U.S. companies are now permitted to export any encryption product around the world to commercial firms, individuals, and other non-government end-users under a license exception (without a license). In addition, "retail" encryption products widely available in the market can now be exported to any end-user including foreign governments. [Ref. 24]

These changes do not include all of the originally proposed measures outlined by the Clinton administration in September 1999; some issues still remain. First, the changes to the U.S. export regulations do not completely eliminate controls on the export of encryption software particularly with regard to the export of binaries. [Ref. 25] Despite these long-awaited changes, there are other barriers to the use of cryptographic software outside of the U.S. such as patent issues for different encryption algorithms.

2. RSA Patent License

Because of RSA intellectual property restrictions and the continued presence of proprietary code licensed from RSA Data Security, Inc., many freeware distribution sites are reluctant to post any source code reliant on the RSA encryption algorithm.

The RSA public key cryptosystem was developed at MIT, and now RSA Data Security, Inc., holds a patent on it. MIT distributes a freeware version of PGP under the terms of the RSAREF license. All versions of PGP use RSAREF, a software library that implements the RSA cryptography routines. Everyone in the U.S who wants to make use of RSA in their programs and distribute it as freeware, must use RSAREF. However, the freeware cannot be used for commercial purposes without purchasing the commercial license.

3. Other Licenses

A conventional block cipher algorithm called IDEA is used by PGP, the SSLeay freeware, and other security applications. Again, this poses a problem for freeware distribution sites since the algorithm is covered by a patent in Europe and inside the U.S. There is no license fee required for non-commercial use of IDEA but commercial users require the purchase of a license.

4. PGP and the International Version

If PGP utilizes RSAREF, then how can we have an international version that is fully compatible with the U.S. version? The answer is astounding. PGP v2.6.2 source code was illegally exported but once out of the country, it was available for distribution legally. [Ref. 26]

The international version does not use the same RSAREF routines as the U.S. version; instead, it uses MPILIB routines that are functionally the same but generally faster, and are backward compatible with the older signature format from earlier PGP releases. Since the international version uses MPILIB vice RSAREF, it should not be used in the U.S.

5. Alternative Security Implementations

Despite the relaxed export control laws, providing freeware download sites for cryptography is still entwined in legal issues. However, it should not nullify research into alternative applications. An incentive for further investigation is the number of new releases introduced since the original research commenced. The following subsections review some of the previously attempted installations, discuss recent releases, and the impact of implementing such systems.

a. PGP v6.5.1

PGP v6.5.1 was the first security system layered on Bamboo starting in July 1999. The only drawbacks appeared to be the lack of a binary for the SGI Irix operating system and a version for international users. It's functionality was tested with Bamboo on Windows NT and performed flawlessly. Subsequently, the source code was made available on October 31, 1999 on the PGP International website [Ref. 15]. A number of additional releases have been published to correct bugs.

The consequences of an available international release and availability of the source code have positive implications for Bamboo. The advantages are that it employs the latest algorithms including longer keys, improved certificate revocation procedures, and is the most recent commercial version usually resulting in future upgrades and prompt technical support. Network Associates also provides a Key Server specifically for v6.5.x available via freeware or commercial purchase.

The impact of migrating to this version is minimal and both versions (v6.5.x and v2.6.x) can be supported simultaneously. The source code modules in Bamboo would require minor modifications for v6.5.x compatibility. The code modifications would be invisible to the end-user. The only hindrance foreseen is the requirement to establish another Bamboo PGP Public Key Server to handle the larger key sizes. This would require the administration of two key servers. If two versions of PGP are undesirable, then by establishing a grace period for end-users to adopt the new key sizes in PGP v6.5.x, Bamboo can seamlessly migrate to the latest release. Many of the command-line commands are similar in v6.5.x.

b. SSLeay

With recent export law changes and renewed hope for further relaxation, it would appear that establishments may be less hesitant to implement their own freeware Certificate Authorities for the generation and distribution of X.509 digital certificates. With respect to Bamboo, this is the area requiring more thorough research from the legal perspective, and would require thorough analysis prior to migrating to a different certificate type. The advantages of such a shift would be the adoption of the X.509 v3 certificate format that is extensively used in the business world and prevalent in

government PKI systems. [Ref. 27] Additionally, by establishing one's own X.509 CA, the certificates can be issued and managed for the Bamboo community of users. This would enable signed files to be downloaded, authenticated, and integrity checked by various organizations utilizing X.509 certificates inside and outside of the Bamboo community. Recent experimentation with Netscape Object Signing technology in an independent project has proven fruitful and implementing such a scheme in Bamboo appears feasible.

The X.509 v3 digital certificate format is incompatible with the PGP certificate format. Supporting both certificate formats is an option but requires two independent key servers and the CA must be fluent in both versions for performing CA duties. Ideally, the X.509 system could be implemented via a transition period allowing users to generate certificates and learn the specifics associated with X.509. Modifications to the Bamboo security application code would be more extensive than the simple migration to PGP v6.5.x stated above. However, the tradeoff is in the widespread use and rapid acceptance of the X.509 standard by commercial and government entities. From a user's perspective, well-documented procedures for X.509 versus PGP would not inhibit or detract from Bamboo object signing but would enhance the overall functionality.

The list of potential security applications for further review does not end here but continues with each new package introduced to the public. Recently, Mozilla offered several open-source security projects worth investigating. [Ref. 28] With each potential application up for review, questions must be answered prior to attempting a migration to another product. Major issues include but are not limited to:

- Is the application scalable?
- Does it hinder non-users of the security application?
- Is it easy to use?
- Is it interoperable with other security applications? Does it try to achieve a security standard?
- What is the overall impact to users in order to migrate?
- Can two systems be simultaneously supported?

6. Recommendation

To maximize integration with popular PKI systems in use today, it would be prudent to pursue conversion of the current Bamboo security implementation to the X.509 format. Companies have been reengineering their business models to take advantage of the rapid growth in the Internet. Adaptation to the X.509 format is a prerequisite to leverage this growth. The overwhelming acceptance of the X.509 format will most likely lead to a larger user-base than PGP.

Vendors often provide discounts (if not free) to educational institutions for the implementation of software systems. The benefit to the vendor is in the use of the institution's name as an advertisement of successful installation sites. This venue could be an inexpensive means of establishing a commercial Certificate Authority system at the Naval Postgraduate School (NPS). The University of Pittsburgh has implemented VeriSign's OnSite PKI system as a pilot program for securing a wide range of University services; the case study document is available on the VeriSign website. The major selling point for NPS is that it is not only an educational institution but also provides a foothold into a potentially large market of government organizations. A successful implementation at NPS may be viewed as an attractive investment for any vendor. The alternative to a commercial system is the implementation of a freeware CA system such as SSLeay.

In summary, the rapid deployment of inexpensive DSL and other technologies will minimize the bandwidth problems plaguing users today. With these improvements, large file transfers will be commonplace. If the security industry can adopt an interoperable object signing standard for implementation across heterogeneous systems, organizations will be able to confidently and safely transfer, authenticate, and verify the content of such files.

APPENDIX A. USER'S GUIDE FOR BAMBOO OBJECT SIGNING

Appendix A details installation steps for PGP v2.6.2 and illustrates procedures to utilize the object signing technology in Bamboo. Most of the information presented here is also available from the Bamboo distribution under: `bb_dir/utls/index.html` (where `bb_dir` stands for the user-specified Bamboo directory).

A. PGP v2.6.2 INSTALLATION PROCEDURES

For a more comprehensive guide to installation instructions, refer to the *user.doc* file included in the PGP v2.6.2 distribution.

1. Windows Install

- U.S. and Canadian residents may obtain *PGP v2.6.2 for DOS* at: <http://web.mit.edu/network/pgp.html>; others may obtain the international version *PGP v2.6.3i for MS-DOS* at: <http://www.pgpi.org/products/nai/pgp/versions/freeware/dos/2.6.3i>.
- Unzip the downloaded zip file, then unzip `pgp262i.zip` (`pgp263ii.zip` for international version) into directory `c:\pgp26` (user-specified drive).
- Set the following variables (add to *autoexec.bat* file or in System Properties under Control Panel); if setting the variables in *autoexec.bat*, reboot for the changes to take effect.
 - set `PGPPATH=c:\pgp26`
 - set `path=c:\pgp26;%path%`
 - set `TZ=time zone; "PST8PDT"` for Pacific, etc. (see *user.doc* file)
- Type: "pgp" at the command prompt; you should receive verbiage such as: "Pretty Good Privacy(tm) 2.6.2 - Public-key encryption..."; if you do not, then PGP was not installed properly.

2. Unix Install

- U.S. and Canadian residents may obtain *PGP v2.6.2 for Unix* at: <http://web.mit.edu/network/pgp.html>. It is counter-intuitive but the "Software Code Download" section for Unix is also located on the DOS download page after answering the export control questions. Choose the desired Unix software download distribution link to acquire the PGP source code; others

may obtain the international version *PGP v2.6.3i for Unix* at:
<http://www.pgpi.org/products/nai/pgp/versions/freeware/unix/2.6.3i>.

- Unix procedures vary by platform; refer to the *user.doc* file for specific implementations. The following procedures were used for SGI Irix.
- In your home directory, type:
 - `mv pgp262s_tar.gz pgp262s.tar.gz`
 - `mkdir pgp262`
 - `cd pgp262`
 - `mkdir build`
 - `cd build`
 - `cp ../../pgp262s.tar.gz .`
 - `gunzip pgp262s.tar.gz`
 - `tar -xvf pgp262s.tar`
 - `tar -xvf pgp262si.tar`
 - `tar -xvf rsaref.tar`
 - `cd rsaref/install/unix`
 - `make CC=cc RANLIB=true`
 - `cd ../../../../src`
 - `make irix CC=cc`
- Edit your *.login* file (or other initialization file) to set the path to the location of the *pgp* executable (followed by “source *.login*” to take effect); for example:
 - `setenv PGPPATH /worka/smithml/pgp262/build/src`
 - `set path=(/worka/smithml/pgp262/build/src PGPPATH $path)`
- Setup *config.txt* file and PGP document files; inside *../pgp262/build* directory, execute the following:
 - `mv config.txt src/.`
 - `mv doc/pgpdoc*.txt src/.`

- Type: “pgp” at the command prompt; you should receive some verbiage such as: “Pretty Good Privacy(tm) 2.6.2 - Public-key encryption...”; if you do not, then PGP was not installed properly.

B. PGP v2.6.2 COMMANDS

Included in the PGP v2.6.2 distribution are two user’s guides: *pgpdoc1.txt* and *pgpdoc2.txt*. Certain PGP commands from *pgpdoc1.txt* relevant to this implementation are highlighted below ([] indicates optional field):

- `pgp -kg`
- generate your own unique public/secret key pair
- `pgp -kvv [userid] [keyring]`
- view the contents of your public keyring
- `pgp -kxa userid keyfile [keyring]`
- extract (copy) a key from your public or secret key ring in ASCII format to a keyfile
- `pgp -ka keyfile [keyring]`
- add a public key (ASCII format) stored in keyfile to your public keyring
- `pgp -kvc [userid] [keyring]`
- view the fingerprint of a public key
- `pgp -kc [userid] [keyring]`
- view the contents and check the certifying signatures of your public keyring
- `pgp -kd your_userid`
- permanently revoke your own key, issuing a key compromise certificate
- `pgp -kr userid [keyring]`
- remove a key or just a userid from your public key ring
- `pgp -ks her_userid [-u your_userid] [keyring]`
- sign and certify someone else's public key on your public keyring
- `pgp -krs userid [keyring]`
- remove selected signatures from a userid on a keyring
- `pgp -kd userid`
- disable or reenable a public key on your own public keyring

C. ARCHIVING AND SIGNING A MODULE

To archive and sign a module, type “archive” at the command prompt under the Bamboo directory, and select option 3 from the menu. This is the option to “Archive/sign a module or sign a previously archived module”. Salient features of option 3 are divided into two separate discussions. The first part illustrates procedures to archive a module with the option to sign it. The second is to sign a previously archived module that may have zero or more signatures attached.

1. Archiving and Signing a Module

The user enters the *name* of the module to archive and the *file type*. *File type* is one of three types (s=source, o=object, d=data). Upon archive completion, the user has the option to sign the module. If the user decides not to sign the module, it is saved as an unsigned archive module in the '.bamboo/cache' directory. If the user chooses to sign the module, and if there is more than one private key, then all of the private key userIDs are displayed. Upon selection of a private key, the user enters his password to digitally sign the module. The file is again saved in the '.bamboo/cache' directory. Figure A.1 is a sample script of a user signing a module.

hendrix 41% archive

**** Bamboo Archive Module Menu: (select 1-4) ****

(Note: previously archived modules must be in .bamboo/cache dir)

- 1.) Display list of signatures attached to archived module.
- 2.) Remove a signature from an archived module.
- 3.) Archive/sign a module or sign a previously archived module.
- 4.) Exit.

: 3

Enter the [path/]name of the module : myModule

Enter type of file to archive (s=source, o=object, d=data) : o

Module tarred and zipped as:

/worka/smithml/.bamboo/cache/myModule.sgi.bar

Do you desire to sign module with your PGP private key? (y/n) : y

List of user id's:

1. Marlon L. Smith <smithml@cs.nps.navy.mil>

Name (userid) to attach to this archive module:

"Marlon L. Smith <smithml@cs.nps.navy.mil>"

You need a pass phrase to unlock your RSA secret key.

Key for user ID "Marlon L. Smith <smithml@cs.nps.navy.mil>"

Enter pass phrase:

Key for user ID: Marlon L. Smith <smithml@cs.nps.navy.mil>

512-bit key, Key ID B06B9FF5, created 2000/01/25

Signed archive module is:

/worka/smithml/.bamboo/cache/myModule.sgi.bar

Figure A.1 Sample Script of a User Signing a Module.

2. Signing a Previously Signed Archive Module

The user enters the *name* of the archived module to sign; the module must be located in the '.bamboo/cache' directory. The user's private keyring is accessed and, if there is more than one private key, all of the private key userIDs are displayed. Upon selection of a private key, the user enters his password to digitally sign the module. Figure A.2 is a sample script of a user signing a previously signed module.

```
hendrix 43% archive

** Bamboo Archive Module Menu: (select 1-4) **
(Note: previously archived modules must be in .bamboo/cache dir)

1.) Display list of signatures attached to archived module.
2.) Remove a signature from an archived module.
3.) Archive/sign a module or sign a previously archived module.
4.) Exit.
: 3

Enter the [path/]name of the module : myModule.sgi.bar

Module currently signed by:
1. "Marlon L. Smith <smithml@cs.nps.navy.mil>"

Continue? (y/n) : y

List of user id's:
1. John Q. Public <public@cs.nps.navy.mil>

Name (userid) to attach to this archive module:
"John Q. Public <public@cs.nps.navy.mil>"

You need a pass phrase to unlock your RSA secret key.
Key for user ID "John Q. Public <public@cs.nps.navy.mil>"

Enter pass phrase:

Key for user ID: John Q. Public <public@cs.nps.navy.mil>
1024-bit key, Key ID 74440BED, created 2000/01/25

myModule.sgi.bar in /worka/public/.bamboo/cache/ is signed!
```

Figure A.2 Sample Script of a User Signing a
Previously Signed Module.

D. DISPLAY SIGNATURES ATTACHED TO A MODULE

To view signatures attached to a signed module, the '.bar' module must be located in the '.bamboo/cache' directory. After typing the "archive" command, select option 1.

Figure A.3 displays two signatures attached to a signed archive module.

```

hendrix 43% archive

** Bamboo Archive Module Menu: (select 1-4) **
(Note: previously archived modules must be in .bamboo/cache dir)

1.) Display list of signatures attached to archived module.
2.) Remove a signature from an archived module.
3.) Archive/sign a module or sign a previously archived module.
4.) Exit.
: 1

Enter the [path/]name of the module : myModule.sgi.bar

Module currently signed by:
1. "John Q. Public <public@cs.nps.navy.mil>"
2. "Marlon L. Smith <smithml@cs.nps.navy.mil>"

```

Figure A.3 Sample Script of Viewing Signatures Attached to a Module.

E. REMOVING A SIGNATURE ATTACHED TO A MODULE

To remove a signature from a signed module, the '.bar' module must be located in the '.bamboo/cache' directory. Select option 2 from the menu after typing "archive" at the command prompt. All of the signatories will be displayed and the user prompted to select one for removal. Figure A.4 is a script illustrating the removal of a signature.

```

hendrix 3% archive

** Bamboo Archive Module Menu: (select 1-4) **
(Note: previously archived modules must be in .bamboo/cache dir)

1.) Display list of signatures attached to archived module.
2.) Remove a signature from an archived module.
3.) Archive/sign a module or sign a previously archived module.
4.) Exit.
: 2

Enter the [path/]name of the module : myModule.sgi.bar

Module currently signed by:
1. "John Q. Public <public@cs.nps.navy.mil>"
2. "Marlon L. Smith <smithml@cs.nps.navy.mil>"

Select signature to remove from archive module (1 - 2, x to exit): 2

Confirm user ID signature to delete: "Marlon L. Smith <smithml@cs.nps.navy.mil>"
(y/n) : y

Signature removed from: myModule.sgi.bar

```

Figure A.4 Sample Script of Signature Removal from a Module.

F. STORING AND RETRIEVING SIGNED MODULES FROM A SERVER

Use FTP to store signed archive modules at the server for distribution. Modules are stored with name mangling included. FTP is also handy for downloading modules for placement locally in the '.bamboo/cache' directory. For dynamic downloads using the Bamboo kernel, refer to the Bamboo documentation available in the download or by typing "bamboo" at the command line.

G. STORING YOUR PUBLIC KEY ON THE KEY SERVER

Copy your public key to a keyfile in ASCII format using the "pgp -kxa" command depicted above. Cut and paste the key information from the ASCII keyfile into the key server web page (currently at <http://hendrix/~smithml/KeyServer.html> but will eventually migrate to the Bamboo website at Ref. [19]) under the heading *Submit a Bamboo PGP public key to the server*. Depress the *Submit* button and you have added a public key to the server for others to access. Figure A.5 illustrates.

Submit a Bamboo PGP public key to the server

- Here's how to add a Bamboo PGP public key to the server's database:
 1. At the command line, execute: `pgp -kxa userid filename` to extract a public key to a file in ascii format (see PGP user's guide for details).
 2. Using your favorite text-editor, open the file created in previous step and cut-and-paste the ASCII-armored version of your public Bamboo key into the text box.
 3. Press "Submit".
- The key server will process your request immediately. If you like, you can check that your Bamboo key exists using the extract procedure above.

Enter ASCII-armored Bamboo PGP key here:

-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: 2.6.2

mQCNAziQGBkAAAEEMzAUkZabAIJWMEWqSBYmOG6
+E6DdXoW SxaqrBdadInCR63
DTGA6/zYuNcXNxCJly2NwWJeDfcqddV3SVuUd1bBN3DTITL8PhQUh
QNOCYruGVk1
gMUIKQv8NqX8Np7S5H1MQJL3Ok4sh6TOS9V10banoH3aENxKlrN17
KWqPIAAUR
ICdSb2JlcnQgRS4gU21pdGggPHNteXRocGNzLm5wcy5uYXZ5Lm1p
D4=
=BQzh
-----END PGP PUBLIC KEY BLOCK-----

Reset

Submit this key to the Bamboo key server!

Figure A.5 Subset of Key Server Web Page for Storing Key.

H. RETRIEVING A USER'S PUBLIC KEY FROM THE KEY SERVER

Access the key server web page via the key server link above and scroll down to the heading *Extract a Bamboo PGP public key from the server*. Type a portion of the

userID of the desired key and depress the *Do the search!* button. The key server performs a pattern match and returns all of the "hits". Click on any of the returned hits to view the public key information. Upon locating the desired key's information, copy it into a text editor and save with a '.asc' extension. Store the public key on your keyring using the "pgp -ka" command. After key verification, it is ready for use. Figures A.6 and A.7 illustrate.

Extract a Bamboo PGP public key from the server

- Here's how to extract a Bamboo PGP public key:
 1. Select either the "Index" or "Verbose Index" check box. The "Verbose" option also displays all signatures on displayed keys.
 2. Type ID you want to search for in the "Search String" box.
 3. Press the "Do the search!" key.
 4. The server will return a (verbose) list of keys on the server matching the given ID. (The ID can be any valid argument to a `pgp -kv(v)` command. If you want to look up a key by its hexadecimal KeyID, remember to prefix the ID with 0x.)
 5. The returned index will have hypertext links for every KeyID, and every bracket-delimited identifier (i.e. `<smithml@cs.nps.navy.mil>`). Clicking on the hypertext link will display an ASCII-armored version of the listed public key.

Index: ☒ Verbose Index: ☐

Search String:

☒ Show PGP "fingerprints" for Bamboo keys

☐ Only return exact matches

Figure A.6 Subset of Key Server Web Page to Extract Key.

Public Key Server -- Index "smith"

Type	bits/keyID	Date	User ID
pub	512/0481F411	2000/01/26	*** KEY REVOKED *** Marlon L. Smith <smithml@cs.nps.navy.mil>
	Key fingerprint = A1 63 8A 10 17 8A 82 1D CC 3C 6D FC 9F B3 3D 28		
pub	1024/D749EAD5	2000/01/27	Marlon Smith <smithml@cs.nps.navy.mil>
	Key fingerprint = 89 AC FC 9A C4 B7 04 19 A8 44 59 44 91 C4 5B F7		

Figure A.7 Result from Key Server Search.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. BAMBOO PGP PUBLIC KEY SERVER ADMINISTRATOR'S GUIDE

Appendix B details procedures to manage the key server. It serves as a guide for the key server administrator and is not intended for an end-user. The PGP Key Server was created by Marc Horowitz; his thesis is available at: <http://www.mit.edu/afs/net.mit.edu/project/pks/thesis/paper/thesis.html>. The freeware distribution site for the PGP Public Key Server software is located at: <http://www.mit.edu/people/marc/pks>. The key server is interoperable with peer PGP key servers installed worldwide. In the Bamboo implementation, the e-mail capability is disabled to preclude communication between the Bamboo PGP Public Key Server and peer servers.

A. PGP KEY SERVER SOFTWARE INSTALLATION PROCEDURES

For detailed installation instructions, refer to the *Readme* file included in the PGP Key Server distribution; it is written to accomodate different Unix systems. The following procedures were used for key server implementation on an SGI Irix. The distribution in this example was 'untarred' into */worka/smithml/pks-0.9.4*. The reader may substitute his installation directory for */worka/smithml*. Review *pks-0.9.4/man* for comprehensive documentation on the various programs. An overview of the key server system is provided in file *pks-intro.8*.

1. Build

Download the PGP Key Server release 0.9.4 and patch #2 from: <http://www.mit.edu/people/marc/pks> (*pks-0.9.4.tar.gz*). 'Gunzip' and 'untar' (*tar -xvf*) the *pks-0.9.4.tar.gz* file. This will create a *pks-0.9.4* directory containing all of the source code and documentation. Under the *pks-0.9.4* directory, perform:

- *./configure --prefix=/worka/smithml/pks-0.9.4*
- *gmake* (used version 3.77)

2. Install

Incorporate patch #2 into the source code then change to the pks-0.9.4 directory. Execute 'gmake install' to install the distribution.

In order to utilize the web server component of the package, the *pks-commands.html* file was modified to reflect the Bamboo implementation. Figures B.1 and B.2 (Parts 1 and 2) compose the Bamboo PGP Public Key Server web page for storage and retrieval of PGP public keys. This web page is currently located at: <http://hendrix/~smithml/KeyServer.html> (will eventually migrate to the Bamboo website at Ref. [22]) and interfaces with the key server software to access the database.

Bamboo PGP® Public Key Server

Page modified on
Nov 15, 1999

- [Extract a Bamboo PGP public key from the server](#)
- [Submit a Bamboo PGP public key to the server](#)

Extract a Bamboo PGP public key from the server

- Here's how to extract a Bamboo PGP public key:
 1. Select either the "Index" or "Verbose Index" check box. The "Verbose" option also displays all signatures on displayed keys.
 2. Type ID you want to search for in the "Search String" box.
 3. Press the "Do the search!" key.
 4. The server will return a (verbose) list of keys on the server matching the given ID. (The ID can be any valid argument to a `pgp -kv(v)` command. If you want to look up a key by its hexadecimal KeyID, remember to prefix the ID with 0x.)
 5. The returned index will have hypertext links for every KeyID, and every bracket-delimited identifier (i.e. `<smithml@cs.nps.navy.mil>`). Clicking on the hypertext link will display an ASCII-armored version of the listed public key.

Index: ☒ Verbose Index: ☐

Search String:

☐ Show PGP "fingerprints" for Bamboo keys

☐ Only return exact matches

Extract caveats:

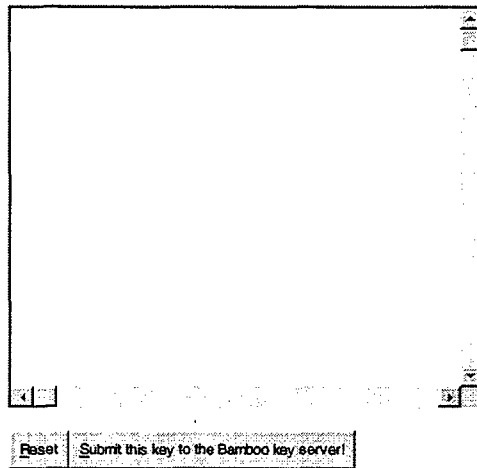
- Hypertext links are only generated for the KeyID and for text found between matching brackets.
- The search engine will return information for all keys which contain all the words in the search string. A "word" in this context is a string of consecutive alphabetic characters. For example, in the string `smithml@cs.nps.navy.mil`, the words are `smithml`, `cs`, `nps`, `navy` and `mil`.
- Keys you may not expect will be returned. Since there may be numerous keys matching either `cs`, `nps`, `navy`, or `mil`, it is prudent to search on `smithml` to reduce the returned matches from the query. If you do not desire all these extra matches, you can select "Only return exact matches", and only keys containing the specified search string are returned.
- This algorithm does *not* match partial words. If you are used to specifying only part of a long name, this does not work.

Figure B.1 Key Server Web Page (Part 1 of 2).

Submit a Bamboo PGP public key to the server

- Here's how to add a Bamboo PGP public key to the server's database:
 1. At the command line, execute: **pgp -kxa userID filename** to extract a public key to a file in ASCII format (see PGP User's Guide for details).
 2. Using your favorite text-editor, open the file created in previous step and cut-and-paste the ASCII-armored version of your public Bamboo key into the text box.
 3. Press "Submit".
- The key server will process your request immediately. Check that your Bamboo key exists using the extract procedure above.

Enter ASCII-armored Bamboo PGP public key here:



The screenshot shows a web browser window with a large text input area for pasting an ASCII-armored PGP public key. Below the text area are two buttons: "Reset" and "Submit this key to the Bamboo key server!". The browser's address bar and status bar are visible at the bottom.

(Thanks to Brian LaMacchia and Marc Horowitz, from whom much of this page is cribbed)
(PGP® is a registered trademark of Pretty Good Privacy)

Figure B.2 Key Server Web Page (Part 2 of 2).

3. Setting Up the Configuration File

The *pksd.conf* configuration file contains all of the necessary parameters for the executable programs of the public key server system. The configuration file is located at *pks-0.9.4/etc/pksd.conf* and is part of the download. Documentation is located in file *pksd.conf.5*. Refer to the documentation for details. The following is a subset of available parameters, and specifically those used for this installation:

- *pks_bin_dir* – Directory location of executables for the key server system (e.g., /worka/smithml/pks-0.9.4/bin).
- *db_dir* – Directory of database files (e.g., /worka/smithml/pks-0.9.4/var/db).
- *www_port* – Port number for accepting HTTP connections (e.g., 11371).

- *help_dir* – Directory of key server help files. The files should be named *pks_help.LANG*, where *LANG* is the language of the help file, in lower-case. (e.g., */worka/smithml/pks-0.9.4/share*)
- *default_language* – Default language English (e.g., *EN*).

4. Running the Key Server

Before running the server, read *pks-intro.8* to familiarize yourself with the Public Key Server System. In the following example, substitute your installation directory for *PREFIX*. (For this particular installation, *PREFIX=/worka/smithml/pks-0.9.4*.)

a. Creating the Database

To create an empty database, execute the following; if a database already exists, then the command overwrites its contents:

- *PREFIX/bin/pksclient /PREFIX/var/dbcreate*

b. Executing the Daemon

To execute the daemon, execute the following command:

- *PREFIX/bin/pksd PREFIX/etc/pksd.conf & sleep 5*

B. PGP KEY SERVER COMPONENTS

The key server system interfaces with the open-source Berkeley Database (DB). The primary components of the key server system are covered here.

1. Server Daemon (*pksd*)

The *pksd* program is the public key server daemon. It processes HTTP requests to add keys to the database and to query the database contents. It is used to add revocation certificates and to disable keys. It reads the *pksd.conf* file for initial parameters. It also receives input from the *pksdctl* program during execution.

2. Server Control Program (*pksdctl*)

The *pksdctl* program is a helper program used by *pksd-mail.sh* and *pksd-queue-run.sh*. (Shells are covered in the documentation.) If the *pksd* program fails unexpectedly, *pksdctl* can be invoked by *pks-mail.sh* and *pks-queue-run.sh* to scan the e-mail transaction queue to update the database. It sends strings to an executing *pksd*. Each string is interpreted as a command. Three commands are supported: *mail*, *disable*,

and *shutdown*, and the command syntax is: *pkscctl* socket string. The *pkscctl* program is not used in this implementation.

- *mail msg* - The file "*msg*" will be parsed as a mail server request. If the file is a valid request, the request succeeds, and the response can be enqueued, then the file is removed.
- *disable userID* - All keys matching "*userID*" are disabled.
- *shutdown* - The key server daemon is shut down.

3. Server Client Program (*pkscclient*)

The *pkscclient* program is a command-line interface for the administrator to perform key server operations directly on the database instead of through the daemon. It supports all of the operations of the daemon and more. The software will use locking and transaction semantics unless specified otherwise.

Each command accepts the directory path of the database files, a command name, and optionally a list of arguments; for instance: *pkscclient /db/path create [num_files]*.

Some commands take an optional flag argument:

- *create [num_files]* - Create an empty database overwriting an existing database if present. The database is split into *num_files* with a default of three.
- *recover* - Recover an inconsistent database. (also see *db_recover*)
- *add filename [flags]* - Add a keyring to the database. The keyring file may be of either '.pgp' or '.asc' format. Figure B.3 below illustrates the *add* function executed under the directory: */worka/smithml/pks-0.9.4/bin*.
 - '*d*' flag - Disabled flag is not stripped from the input file. This is useful when initializing the database for the first time with a keyring from another key server that includes disabled keys.
 - '*t*' flag - Operation takes place without logging and transactions; this is faster, but less safe.

```
hendrix 11% pksclient ./var/db add /worka/smithm/wright.asc
```

```
[Wed Feb 16 14:55:59 2000] kd_open: completed successfully  
[Wed Feb 16 14:55:59 2000] kd_add: flags=100000  
[Wed Feb 16 14:55:59 2000] display_new_key: new keyid 1 A0EBE6C5  
[Wed Feb 16 14:55:59 2000] kd_sync: completed successfully  
[Wed Feb 16 14:55:59 2000] kd_add: pub+1 sig+0 sig=0 uid+0 uid=0 rev+0 rev!0  
[Wed Feb 16 14:55:59 2000] kd_add: completed successfully  
Key block added to key server database.  
New public keys added: 1  
[Wed Feb 16 14:56:00 2000] kd_close: completed successfully
```

Figure B.3 Result of “add filename” Query.

- *get userID [flags]* - An ASCII-armored keyring containing all the keys matching the userID is printed to standard output (stdout). Figure B.4 illustrates a search for the userID just added to the database.
 - ‘e’ flag - The key's userID must be an exact case-insensitive substring of the userID.
 - ‘a’ flag - The userID is ignored, and all keys in the database are returned.
 - ‘b’ flag - The output keyring is in binary format instead of ASCII-armor format.
 - ‘i’ flag - Errors are ignored; this is used when recovering from a corrupt database.
 - ‘d’ flag - Disabled keys are returned.
 - ‘s’ flag - Selected keys are output unsorted to stdout. This flag implies the ‘b’ flag.
 - ‘t’ flag - The operation takes place without logging and transactions; this is faster, but less safe.


```
hendrix 14% pksclient ./var/db get wright
```

```
[Wed Feb 16 15:01:38 2000] kd_open: completed successfully
```

```
[Wed Feb 16 15:01:38 2000] kd_get: userid="wright", flags=0
```

```
[Wed Feb 16 15:01:38 2000] kd_get: completed successfully
```

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
```

```
Version: 5.0
```

```
Comment: PGP Key Server
```

```
mQBNAzirKt0AAAECAK9w1yF6bD7QNkF1U+WQ8e+PG7/w70+iaUKjZc+mbtqQBR0j
```

```
GROLK8iSnWbFCxweRVORKfcZO5YKeigrPqDr5sUABRG0I1RvbSBXcmIlnaHQgPHdy
```

```
aWdodEBjcy5uchMubmF2eS5taWw+
```

```
=Issp
```

```
-----END PGP PUBLIC KEY BLOCK-----
```

```
[Wed Feb 16 15:01:38 2000] kd_close: completed successfully
```

Figure B.4 Result of "get userID" Query.

- *index userID [flags]* - An index listing for all the keys matching the userID is printed to stdout.
 - 'v' flag - Signatures are included in the output.
 - 'f' flag - The key fingerprint is included in the output.
 - 'e' flag - The key's userID must be an exact case-insensitive substring of the userID.
 - 'a' flag - The userID is ignored, and all keys in the database are indexed.
 - 'i' flag - Errors are ignored. This is used when recovering from a corrupt database.
 - 'd' flag - Disabled keys are returned.
 - 's' flag - The index is output unsorted to stdout.
 - 't' flag - The operation takes place without logging and transactions; this is faster, but less safe.
- *since time [flags]* - All keys added to the database or changed since the Unix timestamp are printed to stdout.
 - 'b' flag - The output keys are printed in binary format instead of ASCII-armored format.
 - 'r' flag - The time given is taken as the number of seconds in the past the dump should start.

- 't' flag - The operation takes place without logging and transactions; this is faster, but less safe.
- *delete userID [flags]* - All keys matching the *userID* are deleted from the database.
 - 't' flag - The operation will take place without logging and transactions; this is faster, but less safe.
- *disable userID [flags]* - All keys matching the *userID* have the disabled flag set. Figure B.5 presents an example of a *disable* query executed under */worka/smithml/pks-0.9.4/bin*.
 - 'c' flag - The flag is cleared instead of set.
 - 't' flag - The operation takes place without logging and transactions; this is faster, but less safe.

```
hendrix 8% pksclient ../var/db disable smithml
```

```
[Mon Feb 14 16:26:17 2000] kd_open: completed successfully
[Mon Feb 14 16:26:17 2000] kd_disable: userid="smithml", flags=0
[Mon Feb 14 16:26:17 2000] kd_sync: completed successfully
[Mon Feb 14 16:26:17 2000] kd_disabled: completed successfully
key id 0481F411 disabled
key id D749EAD5 disabled
[Mon Feb 14 16:26:17 2000] kd_close: completed successfully
```

```
hendrix 9% pksclient ../var/db get smithml
```

```
[Mon Feb 14 16:28:06 2000] kd_open: completed successfully
[Mon Feb 14 16:28:06 2000] kd_get: userid="smithml", flags=0
[Mon Feb 14 16:28:06 2000] kd_get: completed with error
database get failed: The requested key has been disabled
[Mon Feb 14 16:28:06 2000] kd_close: completed successfully
```

Figure B.5 Result of “disable userID” Query
followed by “get userID” Query.

C. BERKELEY DATABASE

The open-source Berkeley Database (DB) distribution is available for download at: <http://www.sleepycat.com>. This database is used as the repository for Bamboo PGP public keys. Certain procedures for database management such as recovery, backup, and

checkpointing are briefly discussed here. Comprehensive documentation for this database is available on the website.

1. Checkpointing

Checkpointing the log files commits the database transactions to disk. Transaction commit guarantees that the database changes inside it will never be lost, even after system or application failure. Checkpointing is necessary for two reasons. First, log files can only be removed after a checkpoint. Second, the frequency of checkpoints is inversely proportional to the amount of time it takes to run database recovery after a system or application failure. Once the database pages are written, log files can be archived and removed from the system because they will never be needed for anything other than a catastrophic failure. Additionally, recovery after system or application failure only has to redo or undo changes since the last checkpoint because changes before the checkpoint have all been flushed to the file system.

a. “*db_checkpoint*” Command Description

The *db_checkpoint* utility is a daemon process that monitors the database log and periodically calls another utility, *txn_checkpoint*, for checkpointing.

b. “*db_checkpoint*” Command Syntax

Figure B.6 illustrates a sample “*db_checkpoint*” command execution. The syntax is:

```
db_checkpoint [-l v][-h home][-k kbytes][-L file][-p min]
```

At least one of the *-l*, *-k*, and *-p* options must be specified:

- *-l* Checkpoint the log once, then exit.
- *-h* Specify a home directory for the database.
- *-k* Checkpoint the database at least as often as every *k* bytes of log file written.
- *-L* Log the execution of the *db_checkpoint* utility to the specified file in the following format, where ### is the process ID: “*db_checkpoint*:
Wed Jun 15 01:23:45 EDT 1995”. This file will be removed if the *db_checkpoint* utility exits gracefully.
- *-p min* Checkpoint the database at least every *min* minutes.

- -v Write the time of each checkpoint to stdout.

```
hendrix 29% db_checkpoint -1v -h ./var/db
```

```
db_checkpoint: checkpoint: Wed Feb 16 15:51:58 2000
```

Figure B.6 Example “db_checkpoint” Command.

2. Archiving

Archival is concerned with the recoverability of the database and disk consumption due to the database log files. First, periodic snapshots (i.e., backups) of your databases should be taken in case of catastrophic failure. Second, periodic removal of log files may be necessary to conserve disk space. The two procedures are distinct from each other, and the current log files should not be removed simply because a database snapshot was created.

Log files may be removed at any time as long as they are not involved in an active transaction. By copying the log files to a backup medium before removal, they may be used during restoration of a snapshot to restore your databases to a state more recent than that of the snapshot.

It is often helpful to think of database archival in terms of full and incremental file system backups. A snapshot is a full backup, while the copying of the current logs before removal is an incremental backup. For example, it may be suitable to take a full snapshot of an installation weekly, but archive and remove log files on a daily basis. Using both the snapshot and the archived log files, a crash at any time during the week can be recovered to the time of the most recent log archival, a time later than that of the original snapshot.

a. “db_archive” Command Description

The *db_archive* utility writes the pathnames of log files that are no longer in use (i.e., no longer involved in active transactions) to the standard output, one pathname per line. These log files should be copied to backup media to assist in recovery in case of a catastrophic failure (which also requires a snapshot of the database files). Only then may they be deleted from the system to reclaim disk space.

b. "db_archive" Command Syntax

The *db_archive* command syntax is:

db_archive [-alsv][-h home]

- *-a* Write all pathnames as absolute pathnames, instead of relative to the database home directory.
- *-h* Specify the database home directory.
- *-l* Write out the pathnames of all database log files, whether or not they are involved in active transactions.
- *-s* Write the pathnames of all of the database files that need to be archived in order to recover the database from catastrophic failure. If any of the database files have not been accessed during the lifetime of the current log files, *db_archive* will not include them in this output. It is possible that some of the files referenced in the log have since been deleted from the system. In this case, *db_archive* will ignore them. When *db_recover* is run, any files referenced in the log that are not present during recovery are assumed to have been deleted and will not be recovered.
- *-v* Run in verbose mode, listing the checkpoints in the log files as they are reviewed.

c. Archival for Recovery

The Berkeley DB library supports on-line backups, and it is not necessary to stop accessing the database during the time when the snapshot is created. It is important to note, however, that the snapshot of an active database will be consistent as of some unspecified time between the start of the archival and when archival is completed.

To create a snapshot as of a specific time (recommended procedure for the Bamboo implementation), stop access of the database for the entire time of the archival, force a checkpoint (see *db_checkpoint*), then archive the files listed by the *db_archive* utility's *-s* and *-l* options.

Perform the following steps to create a snapshot of your database that can be used to recover from catastrophic failure:

1. Run `db_archive -s -h` to identify all of the database data files that must be saved, and copy them to a backup device. It may be simpler to archive the whole directory itself instead of the individual files.
2. If reading and writing to the database files while the snapshot is being taken, run `db_archive -l -h` to identify the database log files that must be saved, and copy them to a backup device. Again, consider archiving the directory instead of individual files.

Note that the order of these operations is important, and that the database files must be archived before the log files. This means that if the database files and log files are in the same directory you cannot simply archive the directory; you must make sure that the correct order of archival is maintained.

Once these steps are completed, the database can be recovered from catastrophic failure to its state as of the archival time. To update the snapshot so that recovery from catastrophic failure is possible up to a new point in time, repeat step #2, copying all existing log files to a backup device.

Each time all database and log files are copied to backup media, discard all previous snapshots and saved log files. For archival safety, ensure that you have multiple copies of database backups, preferably on differing storage media.

d. Archival to Conserve Log File Space

To remove log files, take the following steps:

1. If you are concerned with catastrophic failure, first copy the log files to backup media as described above. Log files are necessary for recovery from catastrophic failure.
2. Run `db_archive` without options to identify all of the log files that are no longer in use (i.e., no longer involved in an active transaction).
3. Remove those log files from the system.

3. Recovery

Recovery procedures concern the recoverability of the database. After any application or system failure, there are two possible approaches to database recovery:

- There is no need for recoverability, and all databases can be recreated from scratch. The database home directory can simply be removed and recreated.
- It is necessary to recover information after system or application failure. Recovery (using *db_recover*) must be performed on the database home directory.

Performing recovery will: remove all the shared regions (which may have been corrupted by the failure); establish the end of the log by identifying the last record written to the log; perform transaction recovery. Database applications must not be restarted until recovery completes. During recovery, all changes made by aborted or unfinished transactions are undone and all changes made by committed transactions are redone, as necessary. After recovery runs, the environment is properly initialized so that applications may be restarted. Any time an application crashes or the system fails, recovery processing should be performed on any database environments that were active at that time.

There are two forms of recovery:

1. *normal* or *non-catastrophic* recovery. If the failure is non-catastrophic, (i.e., the database files and log are accessible on a file system that has recovered cleanly), the recovery process will review the logs and database files to ensure that all committed transactions appear and that all uncommitted transactions are undone. Figure B.7 below illustrates a non-catastrophic recovery.
2. *catastrophic* recovery. The Berkeley DB package defines catastrophic failure to be failure where either the database or log files have been destroyed or corrupted. For example, catastrophic failure includes the case where the disk drive on which either the database or logs are stored has been physically destroyed, or when the system's normal file system recovery on startup is unable to bring the database and log files to a consistent state. If the failure is catastrophic, a snapshot of the database files and the archived log files must be

restored onto the system. Then the recovery process will review the logs and database files to bring the database to a consistent state as of the time of the last archived log file. Only transactions committed before that date will appear in the database. Refer to Figure B.8 below.

a. “db_recover” Command Description

The *db_recover* utility is executed to restore the database to a consistent state. All committed transactions are guaranteed to appear after *db_recover* has run, and all uncommitted transactions will be completely undone.

b. “db_recover” Command Syntax

The *db_recover* syntax is:

db_recover [-cv] [-h home]

- -c Failure was catastrophic.
- -h Specify a home directory for the database.
- -v Run in verbose mode.

```
hendrix 28% db_recover -h ./var/db -v
```

```
db_recover: Finding last valid log LSN: file: 1 offset 23946
db_recover: Checkpoint at: [1][23675]
db_recover: Checkpoint LSN: [1][23675]
db_recover: Previous checkpoint: [1][23404]
db_recover: Checkpoint at: [1][23675]
db_recover: Checkpoint LSN: [1][23404]
db_recover: Previous checkpoint: [1][21723]
db_recover: Recovery starting from [1][23404]
db_recover: Recovery complete at Mon Feb 14 17:10:04 2000
db_recover: Maximum transaction id 0 Recovery checkpoint [1][24173]
db_recover: Recovery complete at Wed Dec 31 16:00:00 1969
db_recover: Maximum transaction id 80000000 Recovery checkpoint [1][24173]
```

Figure B.7 Sample Script of “db_recover” Operation.

c. Restoring the Database after Catastrophic Failure

To restore the database after catastrophic failure, perform the following:

1. Restore the copies of the database files from the backup media.
2. Restore the copies of the log files from the backup media.

3. Run `db_recover -c -h` to recover the database. (see Recovery section below)

Figure B.8 is a script illustrating the creation of a database snapshot in case of catastrophic failure: the log files are checkpointed; a snapshot is taken via the archive command; log files are written out to the screen; the database is recovered.

```
hendrix 110% db_checkpoint -lv -h ./var/db

db_checkpoint: checkpoint: Wed Feb 16 17:03:10 2000

hendrix 111% db_archive -s -h ./var/db

keydb000
keydb001
keydb002
keydb003
keydb004
timedb
worddb

(copy database files to backup location)

hendrix 112% db_archive -l -h ./var/db

log.0000000001

(copy log files to backup location)
(after crash occurs, restore backup files)

hendrix 115% db_recover -cv -h ./var/db

db_recover: Finding last valid log LSN: file: 1 offset 77792
db_recover: Recovery starting from [0][0]
db_recover: Recovery complete at Wed Feb 16 17:05:49 2000
db_recover: Maximum transaction id 8000004d Recovery checkpoint [1][77792]
db_recover: Recovery complete at Wed Dec 31 16:00:00 1969
db_recover: Maximum transaction id 80000000 Recovery checkpoint [1][77792]
```

Figure B.8 Database Snapshot and Recovery Procedures.

4. Other DB Features

Deadlock detection and write-ahead logging (DB recoverability) are briefly introduced in this section. The documentation at <http://www.sleepycat.com> is an excellent and detailed resource on these subjects and other topics. Topics such as: file

system operations, logging operations, and utilities: *db_printlog*, *db_load*, and *db_dump* are prerequisites to fully understand the intricacies associated with the Berkeley DB. The two subjects below were selected for discussion, as it is imperative that the administrator be able to recognize deadlock and understand the write-ahead logging feature of the Berkeley DB.

a. Deadlock Detection

Deadlock detection is not so much a requirement specific to transaction protected applications, but rather is necessary for almost all applications where more than a single thread of control will be accessing the database at one time. While Berkeley DB automatically handles database locking, it is normally possible for deadlock to occur.

In order to detect a deadlock, a separate process or thread must review the locks currently held in the database. If a deadlock exists then a victim must be selected. Berkeley DB provides a separate utility, *db_deadlock*, to perform the deadlock detection. Figure B.9 is an example using the *db_deadlock* utility; refer to the website for documentation.

```
hendrix 15% db_deadlock -v -h ./var/db -t 5
db_deadlock: Running at Wed Feb 16 09:49:00 2000
db_deadlock: 3 lockers
db_deadlock: Running at Wed Feb 16 09:49:05 2000
db_deadlock: 3 lockers
```

(Perform recovery here)

```
hendrix 35% db_deadlock -v -h ./var/db -t 5
db_deadlock: Running at Wed Feb 16 10:04:14 2000
db_deadlock: Running at Wed Feb 16 10:04:19 2000
db_deadlock: Running at Wed Feb 16 10:04:24 2000
hendrix 3%
```

Figure B.9 Script of “*db_deadlock*” Operation.

b. Write-ahead Logging

Berkeley DB recovery is based on write-ahead logging. When a change is made to a database page, a description of the change is written into a log file. This description in the log file is guaranteed to be written to stable storage before the changed database pages are written. This is the fundamental feature of the logging system that

makes durability and rollback work. If the application or system crashes, the log is reviewed during recovery. Any database changes described in the log that were part of committed transactions, and that were never written to the actual database itself, are written to the database as part of recovery. Any database changes described in the log that were never committed (i.e., changes related to aborted or unfinished transactions in the log), and that were written to the actual database itself, are backed-out of the database as part of recovery. This design allows the database to be written lazily, and only blocks from the log file have to be forced to disk as part of transaction commit.

This introduction to the Berkeley DB enables the administrator to acquire a general understanding of features inherent in this system. It provides the framework necessary to maintain data consistency required for the end-user. Establishing backup procedures for database recovery and periodically performing these procedures will maximize database accessibility and minimize hardships during recovery evolutions.

LIST OF REFERENCES

1. Rohrbach, M., "Public Key Infrastructure," SPAWAR PMW-161, October 1998.
2. Entrust Technologies, *Entrust Overview*, by C. Voice, 22 October 1998.
3. Entrust Technologies, *Key Update and the Complete Story on the Need for Two Key Pairs*, by I. Curry, December 1998.
4. RSA Security White Paper, *Understanding Public Key Infrastructure (PKI)*, 1999.
5. Naval Postgraduate School, *Bamboo – A Portable System for Dynamically Extensible, Real-time, Networked, Virtual Environments*, presented by K. A. Watsen and M. J. Zyda at the 1998 IEEE Virtual Reality Annual International Symposium (VRAIS'98), Atlanta, Georgia, 14-18 March 1998.
6. Netscape White Paper, "Netscape Object Signing: Establishing Trust for Downloaded Software." [<http://developer.netscape.com/docs/manuals/signedobj/trust/index.htm>]. July 1997.
7. VeriSign White Paper, *VeriSign - Evaluating Enterprise Digital Certificate Solutions*, 1998.
8. VeriSign White Paper, *VeriSign – Guide to Securing Intranet and Extranet Servers*, 1998.
9. Netscape White Paper, "Introduction to Public-Key Cryptography." [<http://developer.netscape.com/docs/manuals/security/pkin/contents.htm>]. October 1998.
10. SSH Communications Security Ltd. White Paper, "Cryptographic Algorithms." [<http://www.ssh.fi/tech/crypto/algorithms.html>]. 2000.
11. Netscape White Paper, "Signing Software with Netscape Signing Tool 1.1." [<http://developer.netscape.com/docs/manuals/signedobj/signtool/signintr.htm>]. June 1998.
12. Machefsky, I., *A Total Economic Impact Analysis of Two PKI Vendors: Entrust and VeriSign*, Giga Information Group, 1998.
13. Network Associates Inc., "An Introduction to Cryptography." [<http://web.mit.edu/network/pgp.html>] included in distribution, November 1998.
14. Schneier, B., "The 1999 Crypto Year-in-Review." [<http://www.infosecuritymag.com/dec99/cryptorhythms.htm>], December 1999.

15. PGP International Website, [<http://www.pgpi.com>]. 1999.
16. Zimmerman, P., *PGP User's Guide Volume I: Essential Topics*, Boulder Software Engineering, 1994.
17. Zimmerman, P., *PGP User's Guide Volume II: Special Topics*, Boulder Software Engineering, 1994.
18. Breed, C., "X.509 vs. PGP Certs." [<http://www.infosecuritymag.com>]. June 1999.
19. Bamboo Website, [<http://www.npsnet.org/~watsen/Bamboo>]. October 1999.
20. Horowitz, M., "PGP Public Key Server." [<http://www.mit.edu/people/marc/pks>]. September 1999.
21. The Berkeley Database Website, [<http://www.sleepycat.com/docs/index.html>]. November 1999.
22. *Webster's II New Riverside University Dictionary*, p. 1240, The Riverside Publishing Company, 1988.
23. Ellison, C., Schneier, B., "Ten Risks of PKI: What You're not Being Told about Public Key Infrastructure," *Computer Security Journal*, Volume 16, pp. 1-7, January 2000.
24. U.S. Department of Commerce Bureau of Export Administration Website, "Commerce Announces Streamlined Encryption Export Regulations." [<http://204.193.246.62/public.nsf/docs/60D6B47456BB389F852568640078B6C0>]. January 2000.
25. Mozilla Crypto FAQ Website, [<http://www.mozilla.org/crypto-faq.html>]. February 2000.
26. Engelfriet, A., "The comp.security.pgp FAQ." [<http://www.uk.pgp.net/pgpnet/pgp-faq>]. October 1998.
27. VeriSign Website, "VeriSign Approved to Provide Digital Certificate Services for U.S. Department of Defense." [<http://www.verisign.com/press/1999/partner/usdod.html>]. October 1999.
28. Mozilla PKI Projects Website, [<http://www.mozilla.org/projects/security/pki>]. 2000.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
 8725 John J. Kingman Rd., STE 0944
 Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
 Naval Postgraduate School
 411 Dyer Rd.
 Monterey, California 93943-5101

3. Prof. Michael Zyda, Code MV 1
 MOVES Academic Group
 Naval Postgraduate School
 Monterey, California 93943-5100

4. Prof. John Falby, Code MV 1
 MOVES Academic Group
 Naval Postgraduate School
 Monterey, California 93943-5100

5. Kent Watsen, Code MV 1
 Naval Postgraduate School
 Monterey, California 93943-5100

6. LCDR Marlon L. Smith, Code MV 1
 Naval Postgraduate School
 Monterey, California 93943-5100